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

MAYLER GAMA ALVARENGA MARTINS

Applications of Functional Composition for CMOS and Emerging Technologies

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

Advisor: Prof. Dr. Andre Inacio Reis
Coadvisor: Prof. Dr. Renato Perez Ribas

Porto Alegre
November 2015

CIP — CATALOGING-IN-PUBLICATION

Gama Alvarenga Martins, Mayler

Applications of Functional Composition for CMOS and Emerging Technologies / Mayler Gama Alvarenga Martins. – Porto Alegre: PGMICRO da UFRGS, 2015.

153 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2015. Advisor: Andre Inacio Reis; Coadvisor: Renato Perez Ribas.

1. Functional Composition. 2. Logic Synthesis. 3. Emerging Technologies. 4. Circuit Resynthesis. 5. Approximate Circuits. 6. Threshold Logic. 7. Majority Logic. 8. Spin-diodes. 9. Memristors. I. Reis, Andre Inacio. II. Ribas, Renato Perez. III. Título.

"I not only use all the brains that I have, but all I can borrow."

THOMAS WOODROW WILSON – 28^{th} PRESIDENT OF THE UNITED STATES

ACKNOWLEDGMENT

Undertaking the Ph.D. was a fantastic journey of learning for me. Completion of this thesis was possible with the support of several people. I would like to express my sincere gratitude to all of them.

Firstly, I would like to say thank you to my advisor, Prof. André I. Reis for the continuous support of my Ph.D. study and related research. I appreciate all his contributions, including ideas and funding that made my Ph.D. so productive regarding publications and conferences. Also, my sincere thanks for my co-advisor Prof. Renato P. Ribas, for all talks, lunches and all the barbecues held at his home.

Besides my advisor and co-advisor, I would like to thank the rest of my thesis committee: Prof. Marcelo Johann, Prof. José Rodrigo Azambuja, and Prof. Paulo Butzen, for their insightful comments.

I thank my fellow labmates in for the fruitful discussions, the last-minute working before deadlines, the paper reviews, and for all the fun we have had.

Also, I would like to thank Dr. Alan Mishchenko for all compelling logic synthesis discussions and networking, and Prof. Jordi Cortadella for accepting me as a visiting scholar in the Universitat Politècnica de Catalunya (UPC).

Last but not the least, I would like to thank my family for all their love and encouragement. For my parents (Luiz C. P. Martins and Maruza G. A. Martins) and my godmother (Ediméa C. Gama), who supported me in everything. Also, by the encouraging and patient support of my girlfriend, Francine B. Puchalski whose dedicated support during the final stages of this Ph.D. is so appreciated. Thank you.

The research presented in the dissertation was supported by Brazilian funding agencies CAPES, CNPq, and FAPERGS, under grant 11/2053-9 (Pronem).

To all that collaborated in this work directly or indirectly, my sincere thanks!

ABSTRACT

The advances in semiconductor industry over the last decades have been strongly based on continuous scaling down of dimensions in manufactured CMOS devices. The use of CMOS devices profoundly relies on AND/OR/Inverter logic. As the CMOS scaling is reaching its physical limits, researchers increase the effort to prolong the CMOS life. Also, it is necessary to investigate alternative devices, which in many cases implies the use of different basic logic operations. As the commercial synthesis tools are not able to handle these technologies efficiently, there is an opportunity to research alternative logic implementations better suited for these new devices. This thesis focuses on presenting efficient algorithms to design circuits in both CMOS and new technologies while integrating these algorithms into regular design flows. For this task, we apply the functional composition technique, to efficiently synthesize both CMOS and emerging technologies. The functional composition is a bottom-up synthesis approach, providing flexibility to implement algorithms with optimal or suboptimal results for different technologies. To investigate how the functional composition compares to the state-of-the-art synthesis methods, we propose to apply this synthesis paradigm into six different scenarios. Two of them focus on CMOS-based circuits, and other four are based on emerging technologies. Regarding CMOS-based circuits, we investigate functional composition to investigate multi-output factorization in a circuit resynthesis flow. Also, we manipulate approximate functions to synthesize approximate triple modular redundancy (ATMR) modules. Concerning emerging technologies, we explore functional composition over spin-diode circuits and other promising approaches based on different logic implementations: threshold logic, majority logic, and implication logic. Results present a considerable improvement over the state-of-the-art methods for both CMOS and emerging technologies applications, demonstrating the ability to handle different technologies and showing the possibility to improve technologies not explored yet.

Keywords: Functional Composition. Logic Synthesis. Emerging Technologies. Circuit Resynthesis. Approximate Circuits. Threshold Logic. Majority Logic. Spin-diodes. Memristors.

Aplicações da Composição Funcional para CMOS e Tecnologias Emergentes

RESUMO

Os avanços da indústria de semicondutores nas últimas décadas foram baseados fortemente na contínua redução de tamanho dos dispositivos CMOS fabricados. Os usos de dispositivos CMOS dependem profundamente da lógica de portas E/OU/INV. À medida que os dispositivos CMOS estão atingindo oslimites fisicos, pesquisadores aumento esforço para prolongar a vida útil da tecnologia CMOS. Também é necessário investigar dispositivos alternativos, que em muitos casos implicam no uso de operações lógicas básicas diferentes. Como as ferramentas comerciais de síntese não são capazes de manipular eficientemente estas tecnologias Esta tese de doutorado foca em produzir algoritmos eficientes para projeto de circuitos tanto em CMOS quanto em novas tecnologias, integrando estes algorithmos em fluxos de projeto. Para esta tarefa, aplicamos a técnica da composição functional, para sintetizar eficiente tanto em CMOS quanto em tecnologias emergentes. A composição funcional é uma abordagem de síntese de baixo para cima, provendo flexibilidade para implementar algoritmos com resultados ótimos ou sub-ótimos para diferentes tecnologias. A fim de investigar como a composição funcional se compara às abordagens de síntese estado-da-arte, propomos aplicar esse paradigma de síntese em seis cenários diferentes. Dois deles se concentram em circuitos baseados em CMOS e outros quatro em circuitos baseados em tecnologias emergentes. Em relação a circuitos baseados em CMOS, investigamos a composição funcional para fatoração de funções multi-saídas, aplicadas em um fluxo de resíntese. Também manipulamos funções aproximadas, a fim de sintetizar módulos de redundância tripla aproximada. No que diz respeito as tecnologias emergentes, exploramos a composição funcional através de diodos spintrônicos e outras abordagens promissoras com base em diferentes implementações de lógica: a lógica de limiar, lógica majoritária e lógica de implicação. Resultados apresentam uma melhoria considerável em relação aos métodos estado-da-arte tanto para aplicações CMOS quanto aplicações de tecnologias emergentes, demonstrando a capacidade de lidar com diferentes tecnologias e mostrando a possibilidade de melhorar tecnologias ainda não exploradas.

Palavras-chave: Composição Funcional, Síntese Lógica, Tecnologias Emergentes, Resíntese de Circuitos, Circuitos Aproximados, Lógica de Limiar, Lógica de Majoritárias, Diodos spintrônicos, memristores.

LIST OF ABBREVIATIONS AND ACRONYMS

AIG	And-Inverter Graph
AOI	And-Or-Inverter
ASIC	Application Specific Integrated Circuits
AST	Abstract Syntax Tree
ATMR	Approximate Triple Modular Redundancy
CAD	Computer-Aided Design
CMOS	Complementary Metal-Oxide-Semiconductor
DAG	Directed Acyclic Graph
DSD	Disjoint Support Decomposition
ECAD	Electronic Computer-Aided Design
EDA	Electronic Design Automation
FATMR	Full Approximate Triple Modular Redundancy
FC	Functional Composition
FET	Field Effect Transistor
FPGA	Field-Programmable Gate Array
GP	Genetic Programming
IC	Integrated Circuit
ISOP	Irredundant Sum-of-Products
IPOS	Irredundant Product-of-Sums
ITRS	International Technology Roadmap for Semiconductors
MDC	Minimum Device Chain
MOS	Metal Oxide Semiconductor
MTJ	Magnetic Tunneling Junction
NMOS	N-type metal-oxide-semiconductor

PI	Primary Input
PLA	Programmable Logic Array
PMOS	P-type metal-oxide-semiconductor
PO	Primary Output
POS	Product of Sums
PPAC	Performance, Power, Area and Cost
QCA	Quantum Cellular Automata
ROBDD	Reduced Ordered Binary Decision Diagram
RPO	Read-Polarity Once
RTD	Resonant Tunneling Diode
RTL	Register Transfer Level
SEE	Single Event Effects
SET	Single Electron Tunneling
SET	Single Event Transient
SEU	Single Event Upset
SOP	Sum of Products
STA	Static Timing Analysis
STTMTJ	Spin Transfer Torque Magnetic Tunneling Junction
TLF	Threshold Logic Function
TLG	Threshold Logic Gate
TMR	Triple Modular Redundancy
TPL	Tunneling Phase Logic
VLSI	Very Large Scale Integration

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

The history of the electronics industry started with Jack Kilby's invention of the first integrated circuit (IC) in 1958, patented in 1963 (KILBY, 1963). Seven years later, Gordon Moore observed that the number of transistors per square centimeter on integrated circuits (ICs) doubled every year since the invention of the IC. His subsequent prediction that this doubling would continue to occur at approximately 18-month intervals has become known as Moore's Law (MOORE, 1965). It is expected that this trend continues for the next 10 years or even longer, mainly for digital circuits (ITRS, 2015).

As the microelectronics industry imposed Moore's law as the ultimate goal, they also needed to adapt to the upcoming challenges, since the design teams are also necessary to increase proportionally to follow the associated complexities in the design of circuits. As this growth in the number of engineers turns to be not feasible in some moment, the computer-aided design (CAD) industry was created to automatize the process and reduce the number of human errors. The scaling integration of devices achieved in the last decades due the heavy use of CAD tools brought positive impact on several aspects. For instance, they allowed reduction of the circuit power consumption per computation, increased the operation frequency and allowed larger integration of devices in the same die. To continue the miniaturization trend (also referred as *scaling*) of the devices, research on synthesis' algorithms for digital systems are necessary.

The digital systems design flow on application-specific integrated circuit (ASIC) is usually divided into two subgroups: logic synthesis, and physical synthesis. The logic synthesis is the process of transforming an abstract form of desired circuit behavior into a design implementation regarding logic components. The circuit behavior is usually expressed utilizing a Register Transfer Level (RTL) description while the design implementation regarding logic components is a logic gate netlist. Logic synthesis involves the abstraction, representation, manipulation, transformation, analysis, and optimization of logic circuits (MICHELI, 1994). The physical synthesis follows up after the end of the logic synthesis, transforming circuit representations of the components into geometric representations of shapes with multiple layers of materials. Physical synthesis steps include floorplanning, placement, routing, clock tree synthesis and others (ALPERT; MEHTA; SAPATNEKAR, 2008). Since physical synthesis is beyond the scope of this work, we will focus on logic synthesis methods.

1.1 Motivation and Challenges

Since 2000, the International Technology Roadmap for Semiconductors (ITRS) is pointing directions for research and industry. Research efforts have been made to enable newer technologies, to continue the advances both in Moore's law and in Koomey's law, which refers the number of computations per joule of energy dissipated (KOOMEY et al., 2011). From all working groups on ITRS, two are of main interest in this thesis: the "More Moore" and "Beyond CMOS" groups. The former group is focused on the continued shrinking of CMOS, focusing on the PPAC characteristics (Performance, Power, Area, Cost). Also, they are interested in improving the reliability of the circuits. The latter investigates devices that are not CMOS based, such as Quantic devices, spintronics, memristors, and others (ITRS, 2015). One of the main objectives of the "More Moore" is extending the CMOS life for more 5-10 years while the "Beyond CMOS" is trying to find one device that is more efficient/cheaper than actual CMOS technology. However, extending CMOS life is a difficult task since scaling CMOS technology, especially into the nanometric regime requires dealing with additional challenges, including not only deep-submicron physical effects but also deep sub-wavelength, lithography limitations, and others. These challenges reduce manufacturing yield, threatening the practicability of smaller CMOS devices (CHOI et al., 2001; PACHA et al., 2006; YONEDA et al., 2008; BORKAR, 2009; KUHN, 2012; MARTINS et al., 2015).

In typical CMOS logic synthesis flows, Boolean expressions are usually written using the AND/OR/INV operators and logic synthesis algorithms aim to optimize such expressions to obtain better circuit implementations in a technology independent manner. The usual technology-independent CMOS logic optimizations can be done by minimizing literals in factored forms (BRAYTON, 1987), or more recently by reducing nodes in an And-Inverter Graph (AIG) representation (MISHCHENKO; CHATTERJEE; BRAYTON, 2006) of a circuit. For instance, optimizations based on literal minimization can be obtained through the minimization of two-level expressions (e.g. sum-of-products - SOP) and the factorization of multi-level Boolean expressions (SENTOVICH et al., 1992). Current state-of-the-art logic synthesis is based on minimization of the logic depth and the total number of nodes of an AIG representation of the circuit (Berkeley Logic Synthesis and Verification Group, 2013). Notice that these technology-independent optimizations are based on a structure where the number of literals or the number of AND2 nodes is minimized. The independence in the optimization is referred to the absence of static pre-characterized CMOS library. However, these representations yet implicitly favor CMOS technology because the nature of the structures is based on operations that can be easily

implemented in CMOS.

Due to all inherent difficulties, new devices have been investigated as promising candidates to replace MOS transistor while the scaling is not physically and/or economically unfeasible. It is important to notice that the basic devices affect the way logic is implemented. The use of CMOS devices relies profoundly on AND/OR/Inverter logic. The availability of alternative devices many times implies the use of different basic logic operations, bringing the need to research alternative logic implementations better suited for the new devices. One of the research areas involve threshold logic which can be applied to resonant tunneling diodes (RTD) (PETTENGHI; AVEDILLO; QUINTANA, 2008) and spin transfer torque magnetic tunneling junction (STT-MTJ) (PATIL et al., 2010) since primitive gates of both technologies are better implemented using threshold gates. A subset of threshold logic is the majority logic, where quantum cellular automata (QCA) (LENT et al., 1993), tunneling phase logic (TPL) (FAHMY; KIEHL, 1999) and single electron tunneling (SET) (AVERIN; LIKHAREV, 1986) have the majority (and minority) voter and inverter as primitive logic elements. Another research direction involves the memristor device and associated logic. The memristor is a two-terminal passive device first theorized by Chua (CHUA, 1971) as a basic circuit element in nonlinear circuit theory. Finally, the magnetoresistive spin-diode has been proposed (FRIEDMAN et al., 2012a), and the possibility to implement a logic family using only diodes has been demonstrated (FRIEDMAN et al., 2012b). It is the first diode based logic family that does not require any transistors. This new device is expected to improve the circuit operation frequency while reducing area and power consumption.

All these technologies differ from CMOS design in which (N)AND, (N)OR, and INV gates represent the primitive elements. For any emerging technology be successful, at least three requirements are necessary. First, the technology by itself must present some characteristics that represent improvements over traditional CMOS transistors. These can be related to higher frequency operation, smaller power consumption, smaller area, and so on. Second, the knowledge used in designing and fabricating CMOS circuits should be straightforwardly adapted to the new technology to make the migration process easier. This is important to avoid a time-consuming learning curve that would alienate potential users. Third, design tools that allow efficient use of the new logic primitives are necessary to extract most benefits that these new technologies have to offer.

One of the ways to improve CMOS and perform efficient synthesis in emerging technologies is applying the Functional Composition (FC) paradigm (MARTINS; RIBAS; REIS, 2012; MARTINS; RIBAS; REIS, 2012). It is a synthesis approach that performs a bottom-up

association of Boolean functions, opposed to the top-down functional decomposition strategy. By performing a bottom-up process, FC has a better control of the implementation cost of the final function.

Functional composition is based on the following principles: (1) representation of logic functions as a bonded pair of functional/structural representations; (2) it starts from a set of initial functions; (3) simpler functions are associated to create more complex functions; (4) a partial order that allows the use of dynamic programming; (5) a set of allowed functions is maintained to reduce execution time/memory consumption.

The functional composition approach is flexible; i.e., it can be configured to provide new alternatives to already known logic synthesis algorithms. FC can provide reduced costs (where costs can be metrics as literals or number of gates) due to the use of a bottom-up approach, where the implementation costs of all subfunctions generated during the synthesis process are known. This method can be used for both CMOS circuits (which the minimization in the independent technology step is the number of literals or logic depth, and the operations are usually AND/OR) as emerging technologies (by composing functions using custom associations, representing basic gates from these technologies).

1.2 Objective

The objective of this thesis is to apply the functional composition both to improve CMOS circuits and synthesize the mentioned emerging technologies efficiently. In this thesis, two FC applications are presented for CMOS. The first application performs a multi-output factorization that is applied in a resynthesis flow which allows improving overall circuit area after the technology mapping. The second algorithm generates approximate circuits, which are used in the triple modular redundancy (TMR) strategy, which consists of triple the circuits to improve reliability. Approximate circuits reduce considerably the overhead imposed by the TMR, sacrificing some protection as a trade-off. Also, 4 FC applications for emerging technologies are presented: synthesis of threshold logic for RTDs; majority logic for QCA, SET, and TPL; spin-diodes and memristors. Due to the mentioned properties of the functional composition, it is expected considerable improvement over previous methods, both for CMOS and emerging technologies.

1.3 Thesis Organization

The next chapters are organized as follows:

Chapter 2: *CMOS and Beyond Overview* — discusses some challenges on CMOS which increased considerably in the new technology nodes. Also, the chapter presents a set of the main emerging technologies that are threshold-logic and majority-logic based, spin-diodes and memristors. For each technology, it is discussed the inherent challenges to perform logic synthesis on them.

Chapter 3: *Logic Synthesis Overview* — provides a logic synthesis overview and concepts, that are used are extensively used in Chapter 5 and Chapter 6.

Chapter 4: *Functional Composition Overview* — depicts an overview of the functional composition and its theory. This chapter discusses the principles that compose the FC methodology. Also, it presents the concept of cost for FC and discusses the dynamic programming concept, which plays a vital role in the FC idea and analyses similar approaches as genetic programming.

Chapter 5: *CMOS Applications Using Functional Composition* — shows 2 applications of FC to CMOS technology. The first is a multi-output factorization algorithm, which is applied in a resynthesis flow, to reduce the area of a mapped circuit. The second algorithm generates approximate functions, which can be used in the triple modular redundancy (TMR) circuits to reduce the overhead area considerably, sacrificing some of the protection.

Chapter 6: *Emerging Tech. Applications Using Functional Composition* — proposes 4 applications using FC for emerging technologies. The first application treats threshold-logic based circuits, which comprise mainly RTD and STT-MTJ technologies. The second one synthesizes majority-logic based circuits, which includes QCA, SET, and TPL technologies. Finally, the third and fourth ones comprise synthesis of spin-diodes and memristors, respectively.

Chapter 7: *Conclusions* — presents the main conclusions, and summarizes the contributions of this work. Also it presents future works for new applications and optimization opportunities in the presented algorithms.

2 CMOS AND BEYOND OVERVIEW

Transistor scaling is a major contributor toward continuous improvement of circuit performance. However, the reduction of transistor dimensions increases fabrication and design challenges, including variations and power densities that threaten to impede CMOS scaling. As a result, much effort has been put forth to develop new devices that may allow further progress in computation capability (ITRS, 2015).

The possibility to utilize algorithms and methodologies employed in CMOS design is desired for new technologies. However, existing algorithms designed for CMOS may not be able to consider the particularities of each technology efficiently. Therefore, several works have discussed algorithms and design strategies focusing on a particular technology.

In this chapter, we will present some challenges of CMOS in the next years, and we will also do an overview of some promising emerging technologies and the respective logic designs. Readers with knowledge of a presented technology can skip the sections without compromising the understanding of the contents discussed in the next chapters.

2.1 CMOS Challenges

As the CMOS devices are smaller, many of the metric effects are becoming more severe for each generation: yield is reducing considerably; the variability in the chip, within-chip, and device-to-device is increasing; the transistors have more leakage among other effects (DENNARD, 2015). In the high level, the logic synthesis' tools need to handle multi-million gates, trying to deliver a physical-aware netlist and the physical synthesis' tools need to route kilometers of wires in a small space, perform an efficient clock distribution (which is more and more difficult due to the increase of registers). Also, electrical characteristics need to be taken into account, due to a lot of effects as Negative and positive-bias temperature instability (NBTI/PBTI), time dependent dielectric breakdown (TDDB) hot carrier injection (HCI) and electromigration (BERNSTEIN et al., 2006). The device modeling is also having difficulties due short-channel effects, such as drain-induced barrier, threshold voltage, increased leakage current and degradation of I_{on}/I_{off} (ZHAO; CAO, 2006). We will discuss more two of these problems, and some solutions adopted: the wire length and reliability.

Wires remain a significant challenge in the design of digital IC. In the previous generations, improvements on the density were achieved by the scaling of the transistors. For memories and caches, the delay present in the wires posed a significant obstacle for further optimization.

Also, it is measured that the delay from the gates reduces and the delay from wires double for each generation. One of the solutions is to use wider wires or repeaters (buffers/inverters). The main problem is that both solutions provide a degradation in energy-per-bit and increases the routing density, turning more difficult the routing task (CALHOUN et al., 2008). Another potential solution is to improve logic synthesis methods to reduce the area furthermore, increasing the number of complex cells and reducing both the number of wires present and the wirelength in the circuit (AMARU et al., 2015a; AMARU et al., 2015b).

Also, reliability will be one of the most significant challenges during the following years (ITRS, 2015; GIELEN et al., 2008). The variability (BERNSTEIN et al., 2006) is one of the most important factors for the yield reduction due to random and systematic errors in a circuit. Also, single event effects (SEE) caused by ionizing particles are the primary source of soft errors in space applications (BAUMANN, 2005). Ionizing particles induce single event transients (SET) in combinational ICs, this may lead to loss of data or functionality. SETs become a larger treat with scaling to the nanoscale technologies due to effects of single event multiple transients. Solutions for the variability include post-fabrication calibration and performance monitors. The TMR proposed by von Neumann (NEUMANN, 1956) remains the most popular fault tolerance technique.

2.2 Emerging Technologies

In this section, we will present some of emerging technologies: RTD, STT-MTJ, QCA, SET, TPL, spin-diodes, and memristors.

2.2.1 RTDs and STT-MTJ Devices

The resonant tunneling diode (RTD) can be considered the most mature type of quantum devices, which are used in high-speed and low-power circuits (AVEDILLO; QUINTANA; ROLDÁN, 2006; PETTENGHI; AVEDILLO; QUINTANA, 2008; CHOI et al., 2009). They operate at room temperature and have a III–V large scale integration process (LITVINOV et al., 2010). The incorporation of RTDs into transistor technologies offers the opportunity to improve the speed and compactness of large scale integration. RTDs exhibit a negative differential resistance (NDR) region in their current–voltage characteristics, which can be exploited to increase the functionality implemented by a single gate significantly. It reduces the

circuit complexity in comparison to conventional MOS technologies (AVEDILLO; QUINTANA; ROLDÁN, 2006).

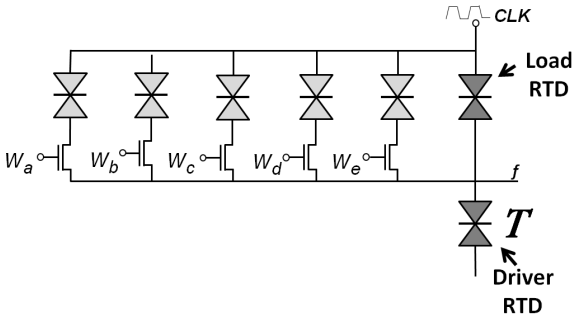


Figure 2.1 – TLG implemented using RTDs (SILVA, 2014).

Another interesting architecture is based on the integration of conventional MOSFETs and a Spintronic device, known in the literature as Spin Transfer Torque - Magnetic Tunneling Junction (STT-MTJ) device (NUKALA; KULKARNI; VRUDHULA, 2014). The novel feature of this architecture is that the STT-MTJ device is intrinsically a primitive threshold device, i.e., it changes its state when the magnitude of the current through the device exceeds some threshold value. This simple property, when exploited, leads to a simple realization of a threshold gate.

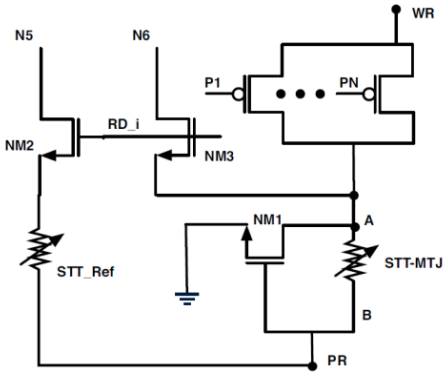


Figure 2.2 – TLG implemented using STT-MTJ (SILVA, 2014).

Recent works discuss the usage of STT-MTJ in logic computation, such as (ZHAO; BELHAIRE; CHAPPERT, 2007; PATIL et al., 2010; GANG et al., 2011). Logic operation performance with STT-MTJ was compared to their traditional methods of computation with separate logic and memory units in (PATIL et al., 2010). All previous works on STT-MTJ for logic use it for storage (logic ‘0’ or ‘1’) or as resistive networks to perform single logic gates. In contrast, the method described in (NUKALA; KULKARNI; VRUDHULA, 2014) employs a single STT-MTJ device in conjunction with MOSFETs to build complex threshold function and is illustrated in Figure 2.2.

Both technologies (RTD and STT-MTJ) can implement logic using threshold gates. These gates are a possible alternative to the Boolean approach. The threshold logic may make possible a considerable economy in the number of gates and interconnections necessary per circuit. Threshold gates are similar to Boolean gates in that their inputs and outputs are binary signals. The threshold gate is thus seen to be a logic function that can "weigh" its various inputs, sum the resultant weighted products, and the output evaluates '1' or '0' if this weighted sum is above or below certain preset threshold values, respectively. This operating behavior can be expressed as Equation (2.1).

$$f = \begin{cases} 1 & \sum_{i=1}^{n} w_i x_i \geq T \\ 0 & \text{, otherwise} \end{cases} \tag{2.1}$$

Threshold logic is closely related to neural networks. Research on this topic dates back sixty years ago. The pivotal year for the development of this area was in 1943 when the first mathematical model of a neuron operating was developed; hence, the threshold logic gate was invented (MCCULLOCH; PITTS, 1943). There were several implementations using CMOS or alternative solutions (LERCH, 1973; BEIU; QUINTANA; AVEDILLO, 2003).

A threshold logic function (TLF), also called linearly separable function, is a Boolean function that can be implemented into a single threshold logic gate (TLG). A TLF can be completely represented in a compact vector format such as $[w_1, w_2, \ldots, w_n; T]$, using Equation (2.1). For instance, the function $f = x_1 \cdot x_2 + x_1 \cdot x_3$ can be represented as $f = [2, 1, 1; 3]$.

A graphic representation of a TLG will be adopted, writing the threshold value inside the node and the input weights at the edges, as illustrated in Figure 2.3 for the Boolean function $f = [2, 1, 1; 3]$.

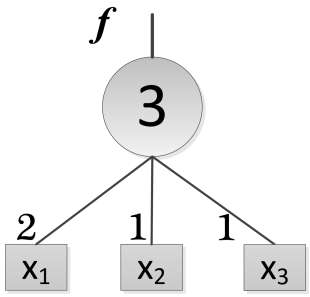


Figure 2.3 – Representation of a threshold logic gate (TLG).

An important property in TLGs is the possibility to implement different Boolean functions only changing the weight inputs and the threshold value. For instance, Figure 2.1 presents a

TLG implemented using RTDs, where each RTD area corresponds to one input weight or the threshold value. Suppose all the input weights be equal to 1 and the threshold value equal to 5. It corresponds to the Boolean function $f = a \cdot b \cdot c \cdot d \cdot e$, the five input AND (AVEDILLO; J.M., 2004). Decreasing the threshold value to 4, the gate implements the function $f = a \cdot b \cdot c \cdot d + a \cdot b \cdot c \cdot e + a \cdot b \cdot d \cdot e + a \cdot c \cdot d \cdot e + b \cdot c \cdot d \cdot e$. Table 2.1 shows different functions can be implemented by keeping all input weights equal to 1 and just decreasing the threshold value.

Table 2.1 – Different functions implemented using different threshold Values in the gate of Figure 2.1

Threshold	Function
t=5	$f = a \cdot b \cdot c \cdot d \cdot e$
t=4	$f = a \cdot b \cdot c \cdot d + a \cdot b \cdot c \cdot e + a \cdot b \cdot d \cdot e + a \cdot c \cdot d \cdot e + b \cdot c \cdot d \cdot e$
t=3	$f = a \cdot b \cdot c + a \cdot b \cdot d + a \cdot b \cdot e + a \cdot c \cdot d + a \cdot c \cdot e + a \cdot d \cdot e + b \cdot c \cdot d + b \cdot c \cdot e + b \cdot d \cdot e + c \cdot d \cdot e$
t=2	$f = a \cdot b + a \cdot c + a \cdot d + a \cdot e + b \cdot c + b \cdot d + b \cdot e + c \cdot d + c \cdot e + d \cdot e$
t=1	$f = a + b + c + d + e$

2.2.2 QCA, SET, and TPL

The electrostatic Quantum-dot Cellular Automata (QCA), proposed by (LENT et al., 1993) employs arrays of quantum cells to implement Boolean functions. This technology has the advantage of an extremely high packing density due the size of dots. In electrostatic QCA, binary information is encoded by the configuration of electrical charges in a QCA cell. Computation is realized via the Coloumbic nature, and current does not flow between cells. Moreover, power dissipation in QCA circuits is considered low compared with conventional CMOS circuits. Also, there are several variations of the QCA technology, as the magnetic QCA (BERNSTEIN et al., 2005) and the molecular QCA (LENT; ISAKSEN; LIEBERMAN, 2003).

The quantum cell consists of four quantum dots located at the corners of the quantum cell and two electrons that can tunnel between the dots. Electrostatic repulsion causes the electrons occupy diagonally opposite sites. These two electron configurations can represent the ‘0’ and ‘1’ binary states. Figure 2.4 shows a QCA cell and its two electron configurations.

The basic QCA logic includes a QCA wire, a QCA inverter, and a QCA majority gate. Figure 2.5 presents the basic QCA elements. A QCA wire (a) is just a line of QCA cells. The information is propagated through a fixed/held polarization and cells in diagonal have reverse polarization. This characteristic is used to implement an inverter (b). A QCA majority gate (c)

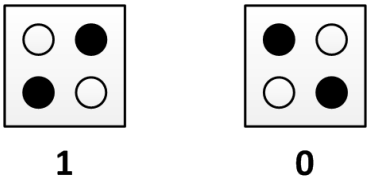


Figure 2.4 – Possible polarizations in QCA cells.

is implemented putting QCA cells in a cross format, where three ends are inputs, and the last one is the output. The logic function that represents a 3-input majority function is described by Equation (2.2).

$$f = x_1 \cdot x_2 + x_1 \cdot x_3 + x_2 \cdot x_3 \tag{2.2}$$

If any of the inputs is set to ‘1’ (‘0’), the majority function implements the OR (AND) logic function. Thus, the majority logic together with the inverter is functionally complete and can realize any Boolean function.

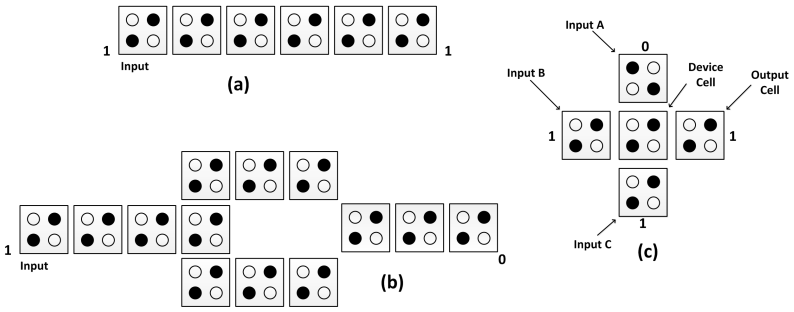


Figure 2.5 – Basic gates in QCA technology.

Also, some technologies use the complement of a majority gate as the basic device. A minority SET gate (AVERIN; LIKHAREV, 1986) is shown in Figure 2.6 (a). It consists of a double-junction box (C_L and two C_j junctions), three input capacitors, and an output capacitor. V_d is the bias voltage. Three input voltages V_1, V_2 and V_3, are applied to Node 1 through the input capacitors. These capacitors form a voltage summing network and produce the mean of their inputs at Node 1. The double-junction box produces the minority-logic output on Node 1 by the following rule. If the voltage at Node 1 exceeds a threshold, an electron will tunnel from the ground to Node 1 via Node 2, and make the voltage at Node 1 negative. Otherwise, the voltage at Node 1 will remain positive. Logic ‘1’ and ‘0’ are represented by a positive and negative voltage of equal magnitude.

A basic minority gate in TPL (FAHMY; KIEHL, 1999) is shown in Figure 2.6 (b). It uses two phases. TPL uses the phase of a waveform to represent logic values in digital circuits.

C_j represents the tunneling junction capacitance. The operation of TPL is based on the phase locking of single electron tunneling oscillations to a pump signal that is distributed throughout the circuit. Because the pump frequency is set to twice the tunneling frequency, the electrical phase of the locked oscillation can take on two different values, each representing a binary encoding.

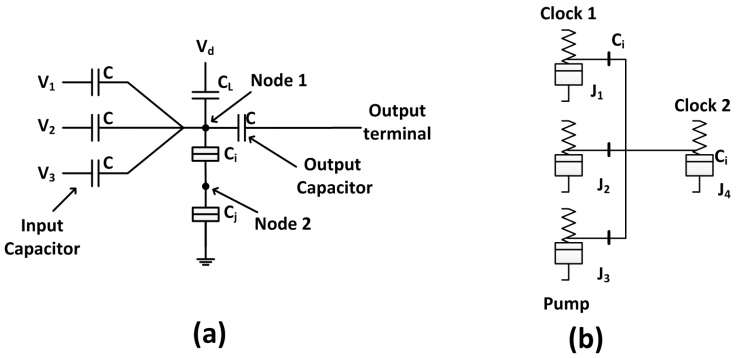


Figure 2.6 – (a) SET and (b) TPL minority gates.

The QCA, SET, and TPL have in common a majority (minority)-based logic. This logic can be considered as a sub-area of the threshold logic synthesis, which dates back to 1960's with the works of Akers (AKERS, 1962), Miller (MILLER; WINDER, 1962), and Muroga (MUROGA, 1971).

2.2.3 Spin-Diodes

The spin-diode is a magnetoresistive p-n junction, i.e., a diode with a resistance that is affected by the magnetic field. For instance, spin-diodes of the type shown in Figure 2.7 have been fabricated by doping a III-V semiconductor heterojunction with an element that has a strong interaction with a magnetic field (MAY; WESSELS, 2006; RANGARAJU; LI; WESSELS, 2009), such as Mn. The spin-diode acts as a conventional diode in the presence of zero or low magnetic fields, with a high ratio of forward current to reverse current. However, when a magnetic field is applied across the junction, spin-dependent conduction results in decreased charge flow across the junction (PETERS et al., 2011).

An inverter, the simplest gate, is shown in Figure 2.8. The positive terminal of the spin-diode is connected to power supply (V_{DD}), and the negative terminal is connected to ground through the output loop. The input current I_A is routed alongside the diode and induces a magnetic field proportional to this current. If I_A is high enough (logic '1'), it creates a large

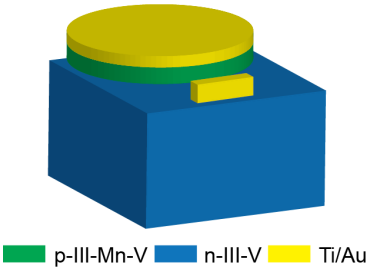


Figure 2.7 – *III-V/III-Mn-V* heterojunction diode (FRIEDMAN et al., 2012b).

magnetic field that reduces the diode current I_O to a value small enough (logic '0'). If I_A is low (logic '0'), the generated magnetic field is not strong enough to cause a decrease of the diode current. Then, I_O is high (logic '1').

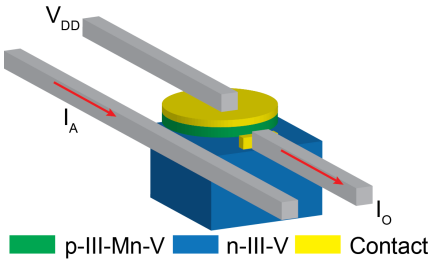
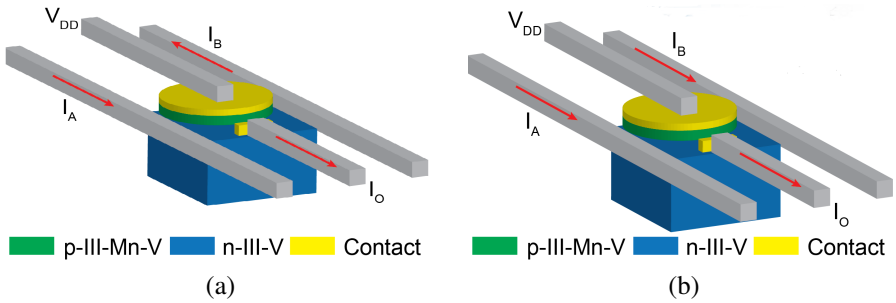


Figure 2.8 – Spin-diode inverter gate (FRIEDMAN et al., 2012b).

An NOR gate is implemented by adding second input current wire I_B in the opposite direction of I_A, as shown in Figure 2.9a. In this arrangement, the presence of a current in either one of the two inputs I_A and I_B results in a magnetic field through the diode. This magnetic field activates the spin-diode magnetoresistance, forcing the diode into the high resistance state and attenuating the current. In the case of high currents on both inputs, the output current is doubly suppressed. Therefore, if at least one of the two inputs is a '1', the output propagates a '0'. Otherwise, there is no magnetic field through the spin-diode, and the output is a '1'. The NOR gate is functionally complete, allowing for the implementation of any Boolean function with these spin-diodes.

The exclusive-NOR (XNOR) gate also requires only one diode. The difference between the XNOR and NOR gates is the current directions in the wires. While the input currents have opposite directions in a NOR gate, in an XNOR gate they flow in the same direction, as seen in Figure 2.9b and the magnetic fields oppose each other. Therefore, if both inputs are '1', there is no net magnetic field through the diode, and a logic '1' value is propagated.

An OR gate can be constructed by simply connecting two wires (wired-OR). As shown in Figure 2.10, the output current I_O is equal to the sum of the two input currents. Even though



(a) (b)

Figure 2.9 – Spin-diode logic (a) NOR and (b) XNOR gate (FRIEDMAN et al., 2012b).

no diodes are required for the wired-OR implementation, this option has the drawback that the input signals are lost. Therefore, it is only possible to use the wired-OR if the input signals are not used elsewhere in the circuit. The same discussion is valid for OR gates with more inputs.

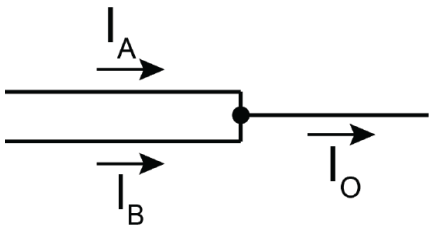


Figure 2.10 – Spin-diode inverter gate (FRIEDMAN et al., 2012b).

2.2.4 Memristors

The memristor can be seen as a two terminal devices with the terminals being separated by a material that is part conducting and part insulator (RAJENDRAN et al., 2012). When the voltages applied to both terminals are different, the electrical properties of the middle material are modified, and the memristor resistance changes. When there is no voltage difference between the terminals, the resistance is not affected. If a positive bias is applied, then the resistance tends to decrease until it reaches a minimum value. Similarly, when a negative bias is applied, the resistance increases until it reaches a maximum value. Figure 2.11a shows the electric symbol for a memristor. Figure 2.11b illustrates the resistance switching for both positive and negative bias.

There are different approaches implementing digital circuits using memristors. Memristors can be used together with traditional CMOS inverters to implement threshold logic gates (RAJENDRAN et al., 2012). In this case, the memristors provide both the input weights and the threshold value, and logic values are defined by voltages. Alternatively, the resistance

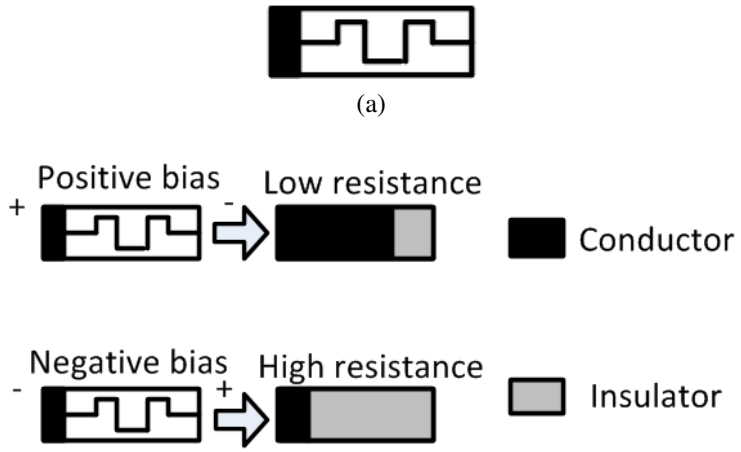


Figure 2.11 – Memristor: (a) electrical symbol and (b) resistance switching due to different bias.

can be exploited to represent logic values (BORGHETTI et al., 2010). In this way, memristors can act both as memory and computing units. One of the main motivations to study this kind of logic is the possible integration with a memristive crossbar memory to realize logic in memory (ZHU et al., 2013). Performing logic in memory can be advantageous since it potentially reduces the number of memory access by the processor, which is a key factor limiting processor performance (WULF; MCKEE, 1995).

The memristor, first idealized as a basic device by Chua (CHUA, 1971), is a two terminal passive device that behaves like a resistor with memory. In (CHUA; KANG, 1976), the concept of memristive devices is presented as an extension of the memristor concept. The terms memristor and memristive devices have been used interchangeably. Even though memristive behavior has been observed for two centuries (PRODROMAKIS; TOUMAZOU; CHUA, 2012), only in 2008 the link between theory and experimental results was made (STRUKOV et al., 2008).

The basic structure to perform implication based logic using memristors is depicted in Figure 2.12, comprising two memristors (P and Q) and a resistor Rg. The state of P and Q are given by p and q, respectively. The logic zero is represented by a memristor in a high resistance state, and the logic one is defined by a memristor in a low resistance state. The memristors P and Q are driven by voltages Vp and Vq, respectively, which can be generated by traditional CMOS tri-state drivers.

Vp and Vq can assume three values: $Vset$, $Vclear$ and $Vcond$. Considering only a memristor in series with Rg, $Vset$ forces the state of the memristor to change to a low resistance state, $Vclear$ forces the state of the memristor to change to a high resistance state and $Vcond$ does not change the state of the memristor.

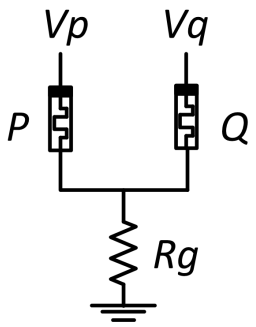


Figure 2.12 – Basic structure to perform implication logic using memristors.

Table 2.2 – Truth table generated by the circuit in Figure 2.12.

p	q	$p \rightarrow q$
0	0	1
0	1	1
1	0	0
1	1	1

In the case when the two memristors are placed in parallel, the voltages Vp and Vq can interact. This interaction is what allows logic to be performed. Consider that $Vset$ is applied to Q at the same time that $Vcond$ is applied to P. If P is in a high resistance state ($p = 0$), then Vp has negligible influence on the circuit and $Vset$ is able to set Q to a low resistance state ($q = 1$). In spite, if p is in a low resistance state ($p = 1$), then Vp increases the voltage across the resistor Rg in such a way that the resulting voltage drop in Q is not enough to cause a change of state and q is not altered. The truth table that represents the performed operation is shown in Table 2.2 and it corresponds to the truth table of the material implication function. Every time an operation $p \rightarrow q$ is executed, the result is stored in q.

In order to implement a NAND operation between P and Q a work memristor ($M1$) is added, the state of $M1$ is given by $m1$ and $M1$ is driven by a voltage $Vm1$. A NAND operation requires three steps. The first step is to initialize R applying $Vclear$ to $Vm1$. The second step is to perform $p \rightarrow m1$, which is the same as $m1$ receives $\neg p$. The third step is to perform $q \rightarrow m1$ which results in $m1 = \neg p \vee \neg q$ (BORGHETTI et al., 2010).

To perform any Boolean function, a second work memristor $M2$ must be added (LEHTONEN; POIKONEN; LAIHO, 2010). $M2$ is driven by a voltage $Vm2$ and its state is given by $m2$. That way, for an arbitrary Boolean function with n inputs $p1, p2, \ldots, pn$, $n + 2$ memristors are sufficient to compute the function. The circuit is shown in Figure 2.13. Memristors $P1$ to Pn store the values of $p1$ to pn, $M1$ and $M2$ are the two working memristors. At any step, one of the working memristors stores the partial result of the computation and the other acts as an auxiliary

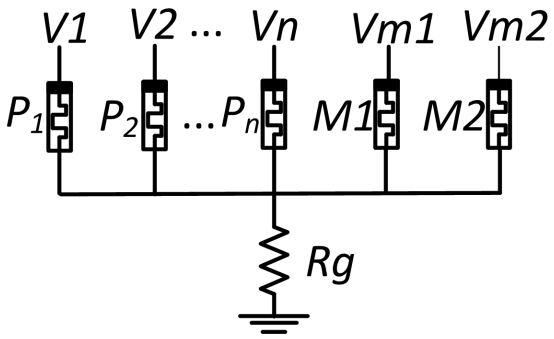


Figure 2.13 – Memristor based implication logic gate.

memristor. The operations allowed are resetting a memristor to a high resistance state and performing a material implication operation between memristors. If the material implication is performed between both working memristor, the roles of the working memristors are exchanged.

2.3 Summary between Emerging Technologies

Table 2.3 shows a summary of all addressed logic from emerging technologies and their primitive gates. The use of CMOS algorithms will have deleterious effects as these algorithms are not optimized for emerging technologies basic gates. We can take advantage of the functional composition to efficiently implement logic for the mentioned emerging technologies. These modifications consist of algorithms for synthesis of functions using corresponding primitive gates that consider the characteristics of the each technology to generate efficient implementations.

Table 2.3 – Summary of technologies and their primitive gates

Technology	Primitive Gates
RTD & STT-MTJ	Threshold Gates
QCA, SET & TPL	Maj/Min Gates
Spin-Diodes	INV/ NOR/ XNOR / OR
Memristor	Material Implication

3 LOGIC SYNTHESIS OVERVIEW

In this chapter, we present an overview of the logic synthesis flow and some important concepts and terminologies commonly used in logic synthesis algorithms. Readers with knowledge in these fields can skip this chapter without compromising the understanding of the contents discussed in the next chapters.

Logic synthesis is the transformation of a circuit behavior description (usually in an abstract form) into a design implementation regarding logic components. This step of the IC synthesis flow connects the high-level synthesis (which generates a register-transfer level from a behavioral description) and physical design (which takes as input a logic gate netlist and provides a placed and routed circuit). Logic synthesis involves the abstraction, transformation, analysis, and optimization of logic circuits. All these operations take place in the transformation from the RTL description to the gate level description (MICHELI, 1994). An important task of the logic synthesis step is to perform automatic generation of logic gates netlists, such that timing constraints imposed by the designer are respected, and the final logic gate netlist has minimized area and power.

Figure 3.1 shows a possible flow for logic synthesis, which is usually divided into technology-independent and technology-dependent steps (MICHELI, 1994). The initial RTL description is parsed into a technology-independent description. The parsing step is considered trivial, and few authors mention about it (MICHELI, 1994). Even so, this parsing is a crucial phase, since few modifications can impact considerably the next steps (PUGGELLI et al., 2011). The output of the parsing step can be a Boolean network [1]. A Boolean network is a graph of connected nodes, where each node represents a Boolean function and connects with other nodes, implementing all combinational logic for the parsed circuit. Figure 3.2 illustrates a Boolean network, where a, b, c, d and e are the primary inputs and G, H are the primary outputs.

The technology mapping (or technology binding) step transforms a logic-independent network into library gates mapped circuit (ASIC) (DETJENS et al., 1987) or look-up tables mapped FPGA (CONG; DING, 1994). In this text, we will focus on ASIC technology mapping. It starts transforming optimized independent logic networks into a subject description, which is a representation of a logic function using only pre-chosen elements (simple gates from a cell library, usually NAND/NOR gates). Usually the subject description is a graph (KUKIMOTO; BRAYTON; SAWKAR, 1998), a tree or a forest of trees (KEUTZER, 1988) or choice

[1] We are considering, for the sake of simplicity, a Boolean network as the output of the parsing step, but it can be any data structure that can represent a circuit, e.g. an And-Inverter Graph (AIG) (MISHCHENKO; CHATTERJEE; BRAYTON, 2006) or an Abstract Syntax Tree (AST) (PUGGELLI et al., 2011).

nodes (MISHCHENKO et al., 2005). A matching is applied in the subject description, which will implement all nodes into cells. This matching can be using patterns (KEUTZER, 1988) in the subject description or taking advantage of Boolean characteristics and properties (MAILHOT; MICHELI, 1990). Finally, a covering step is executed; trying to identify the best matches (using a cost function as area or delay) for the subject description and assuring that every node is covered by a library cell at least once, maintaining the functionality (BRAYTON et al., 1987a; BRAYTON et al., 1987b). One of the pioneering algorithms for technology mapping was the tool DAGON, proposed by (KEUTZER, 1988). The algorithm does the partition of the Boolean network into a forest of trees, applies a pattern matching (representing all library gates as trees) and covers each tree optimally (using dynamic programming). The output is a gate netlist, functionally equivalent to the original RTL description.

Finally, the dependent technology optimization is closely related to the technology mapping, considering in the technology mapping other cost criteria beyond area, such as power and delay (TSUI, 1998), considering timing and power constraints provided by the user. These optimizations are done at same time of technology mapping. The output of this step is a gate netlist attending the design constraints, which will be the input of the physical synthesis. In the next subsections, some basic concepts in logic synthesis will be presented.

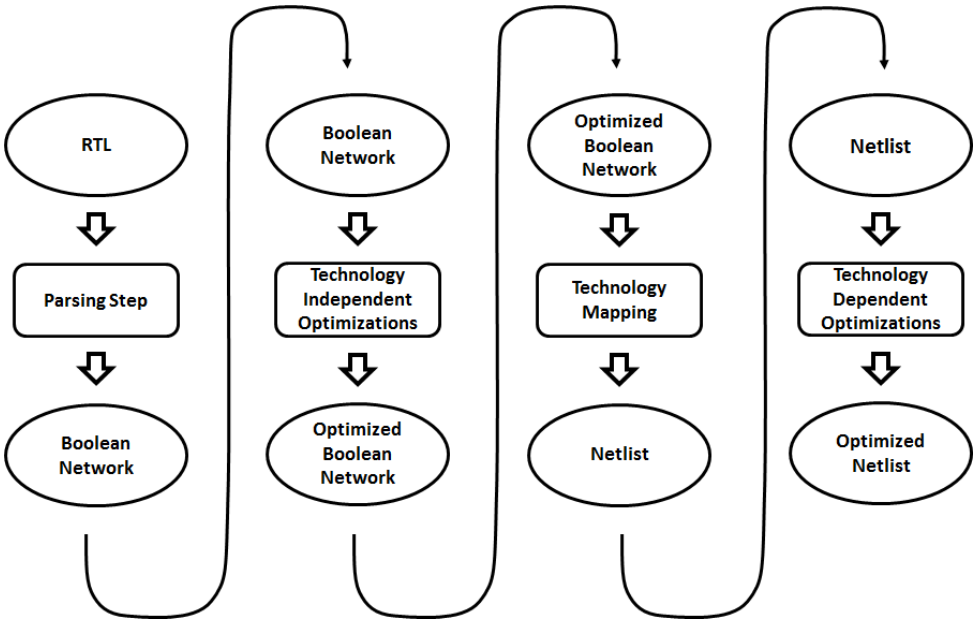


Figure 3.1 – A logic synthesis flow.

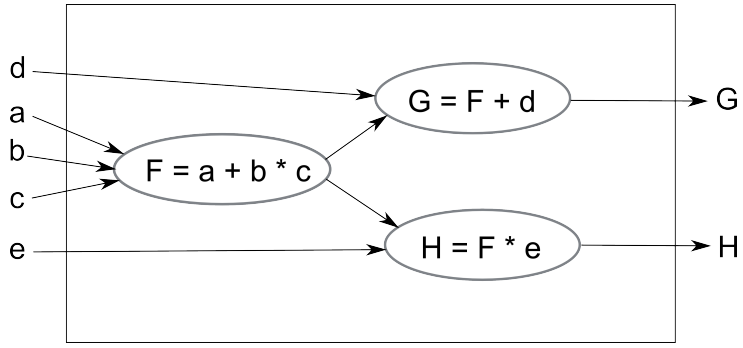


Figure 3.2 – An example of Boolean network.

3.1 Functional Representation and Properties

In this section will be presented the truth table representation of a Boolean function and its functional properties.

3.1.1 Boolean Functions

Let $\mathcal{B} = \{0, 1\}$. A Boolean logic function f with n input variables $[x_1, x_2, \ldots, x_n]$ and one output variable, is a function:

$$f : \mathcal{B}^n \mapsto \mathcal{B} \tag{3.1}$$

where $x = [x_1, x_2, \ldots, x_n] \in \mathcal{B}^n$ is the input of f. This is a representation for a completely specified Boolean function (CSF) taking values from $\mathcal{B}$, i.e., all the values of the input map into 0 or 1 for all components of f. For each function f , it can be defined as follows: the on-set $\mathcal{X}^{ON} \subseteq \mathcal{B}^n$ is the set of input values x such that $f(x) = 1$, and the off-set is the set of $\mathcal{X}^{OFF} \subseteq \mathcal{B}^n$ input values x such that $f(x) = 0$.

3.1.2 Truth Tables

A truth table is one possible representation of a logic function. In this form, the value of the function is specified for each possible combination of inputs. For instance, let $f : x \mapsto y \mid y \in \mathcal{B}$, where the values of $[x_1, x_2, x_3]$ is indicated in Table 3.1.

For this function, we have the on-set and off-set defined respectively by $\mathcal{X}^{ON} = [0, 0, 0], [1, 0, 0], [1, 1, 0]$ and $\mathcal{X}^{OFF} = \{[0, 0, 1], [0, 1, 0], [0, 0, 1], [1, 0, 1], [1, 1, 1]\}$.

Table 3.1 – Truth table for a logic function with 3 variables.

x_1	x_2	x_3	y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

3.1.3 Shannon Expansion and Cofactors

The Shannon expansion (or Shannon decomposition) is defined as (SHANNON et al., 1949):

$$f(x_1, \ldots, x_i, \ldots, x_n) = x_i \cdot f(x_1, \ldots, 1, \ldots, x_n) + \overline{x_i} \cdot f(x_1, \ldots, 0, \ldots, x_n) \tag{3.2}$$

Also, it can be represented in other 2 formats:

$$f(x_1, \ldots, x_i, \ldots, x_n) = x_i + f(x_1, \ldots, 0, \ldots, x_n) \cdot \overline{x_i} + f(x_1, \ldots, 1, \ldots, x_n) \tag{3.3}$$

and

$$f(x_1, \ldots, x_i, \ldots, x_n) = x_i \cdot f(x_1, \ldots, 1, \ldots, x_n) \oplus \overline{x_i} \cdot f(x_1, \ldots, 0, \ldots, x_n) \tag{3.4}$$

The cofactor is a sub element of a Shannon expansion. The Shannon expansion is a way to express a Boolean function by the sum (or product) of two subfunctions of the original. Considering a function f with the input variables $\{x_1, \ldots, x_i, \ldots, x_n\}$, the cofactor f_{x_i} is defined as:

$$f_{x_i} = \{f(x_1, \ldots, x_i, \ldots, x_n) | x_i = \mathcal{B}, k \in \mathcal{B}\} \tag{3.5}$$

The positive cofactor is defined when $k = 1$ and the negative cofactor is defined when $k = 0$. For simplicity, let $f_{x_{i=1}}$ and $f_{x_{i=0}}$ represent positive and negative cofactors, respectively, in the variable of the function f. A cube cofactor is obtained by setting more than one input variable to specific values that can be zero or one (e.g. $f_{x_1=0,x_2=1}$). The cube cofactors are commutative operations.

3.1.4 Unateness

Let f be a Boolean function. The variable x_k in the function f is "don't care" if $f_{x_k=0} = f_{x_k=1}$. The variable x_k in the function f is positive unate if $f_{x_k=0} + f_{x_k=1} = f_{x_k=1}$. The function f is negative unate in the variable x_k if $f_{x_k=0} + f_{x_k=1} = f_{x_k=0}$. Otherwise, the variable x_k in the function f is binate. Let $\mathcal{U}(f, x_k)$ denote the unateness detection function of a variable x_k in the function f, and auxiliary function $i = f_{x_k=0} + f_{x_k=1}$, we have:

$$\mathcal{U}(f, x_k) = \begin{cases} \text{positive unate} & (f_{x_k=1} \equiv i) \wedge (f_{x_k=1} \neq f_{x_k=0}) \\ \text{negative unate} & (f_{x_k=0} \equiv i) \wedge (f_{x_k=1} \neq f_{x_k=0})) \\ \text{don't care} & f_{x_k=1} \equiv f_{x_k=0} \\ \text{binate} & (f_{x_k=1} \neq f_{x_k=0}) \wedge (f_{x_k=1} \neq i) \wedge (f_{x_k=0} \neq i) \end{cases}$$

3.1.5 Symmetry and Antisymmetry

Two or more variables are symmetric when they can be interchanged without modifying the logic function. Two or more variables are antisymmetric if they can be inverted and exchanged to each other without changing the logic function.

For a function $f(x_1, \ldots, x_i, \ldots, x_n)$ with $n \geq 2$, symmetry and antisymmetry of two variables x_i and x_j can be detected comparing the cube cofactors of x_i and x_j . Let $\mathcal{S}(f, x_i, x_j)$ denote the symmetry check of variables x_i and x_j in the function f.

$$\mathcal{S}(f, x_i, x_j) = \begin{cases} \text{symmetric} & f_{x_i=1,x_j=0} = f_{x_i=0,x_j=1} \\ \text{not symmetric} & otherwise \end{cases}$$

The antisymmetric property is similar, changing only the cube cofactors to be checked. Let $\mathcal{AS}(f, x_i, x_j)$ denote the antisymmetry check of variables x_i and x_j in the function f.

$$\mathcal{AS}(f, x_i, x_j) = \begin{cases} \text{antisymmetric} & f_{x_i=0,x_j=0} = f_{x_i=1,x_j=1} \\ \text{not antisymmetric} & otherwise \end{cases}$$

3.1.6 Self-Dual Functions

The dual of a function $f(x_1, x_2, \ldots, x_n)$ is the function $f^d = \overline{f(\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n})}$. Notice that the function f^d is obtained first by replacing each x_i with $\overline{x_i}$ and then complementing the function f. A self-dual function is a function such that $f = f^d$. For instance, in Table 3.2 is presented a self-dual function, called f, which implements the three-input majority function.

Table 3.2 – Truth table of self-dual function f.

Line	x_1	x_2	x_3	f
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

The values in the line 0, 1, 2 and 3 have complemented values of the lines 7, 6, 5 and 4, respectively. This characterizes a self-dual function.

3.1.7 Order

Two Boolean functions can be compared and classified according to their relative ordering, which can be equal, larger, smaller, not-comparable or disjoint. Let $\mathcal{O}(f, g)$ denote the order of f against g, and h be the auxiliary function $h = f + g$, we have:

$$\mathcal{O}(f,g) = \begin{cases} \text{equal} & f = g \\ \text{smaller} & (h = g) \wedge (f \neq g) \\ \text{larger} & (h = f) \wedge = (f \neq g) \\ \text{not comparable} & (f \neq g) \wedge (f \neq h) \wedge (g \neq h) \wedge (f \cdot g \neq 0) \\ \text{disjoint} & (f \neq g) \wedge (f \neq h) \wedge (g \neq h) \wedge (f \cdot g = 0) \end{cases}$$

The order of two functions can be easily observed in a Karnaugh map, shown in Figure 3.3. The shaded area represents the minterms of f projected in g.

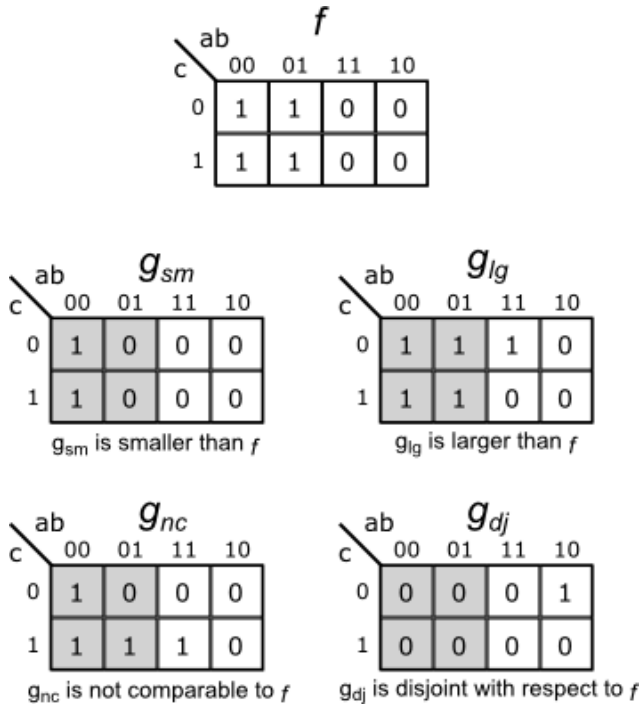


Figure 3.3 – Order visualized in the Karnaugh map.

3.1.8 Boolean Operations

Four basic Boolean operations are discussed in this section. These operations are illustrated in Figure 3.4 and the truth tables representing the operations are shown in Table 3.3.

The complement or negation (NOT, $\neg$) of a logic function f is the logic function $\overline{f}$, where $\overline{f}_{on} = f_{off}$ and $\overline{f}_{off} = f_{on}$.

The intersection or product (AND, $\cdot$) of two logic functions f and g, $h = f \cdot g$ is defined to be the logic function h, where $h_{on} = f_{on} \cap g_{on}$.

The union or sum (OR, $+$) of two logic functions f and g, $i = f + g$ is defined to be the logic function i, where $i_{on} = f_{on} \cup g_{on}$.

The symmetric difference or exclusive sum (XOR, $\oplus$) of two logic functions f and g, $j = f + g$ is defined to be the logic function j, where $j_{on} = (f_{on} \cup g_{on}) \setminus (f_{on} \cap g_{on})$.

Table 3.3 – Truth tables for the following operations: negation, product, sum and exclusive product operations, respectively.

NOT(f)		AND(f, g)			OR(f, g)			XOR(f, g)		
		f	g	h	f	g	i	f	g	h
f	$\overline{f}$	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	1	1	0	1	1
1	0	1	0	0	1	0	1	1	0	1
		1	1	1	1	1	1	1	1	0

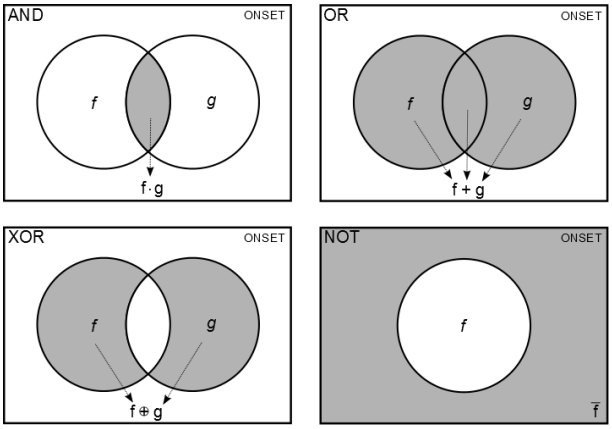


Figure 3.4 – Boolean operations: the grey area represents the result of each operation.

3.2 Boolean Equations

An algebraic representation of f is a Boolean expression that evaluates to 1 for all inputs in $\mathcal{X}^{ON}$ and evaluates to 0 for all inputs in $\mathcal{X}^{OFF}$. An algebraic representation of f can be built by inspection from the truth table of f. For instance, the algebraic representation of f can be constructed as follows. Consider every row of the truth table that has a 1 in the output value. Create a Boolean product (logical "and", represented by operator $\cdot$) of the n input variables $[x_1, x_2, \ldots, x_n]$, the variable x_j appears complemented if the corresponding value of the input variable in the row of the truth table is 0 and direct if it is 1. This product evaluates to 1 for the input combinations corresponding to the row of the truth table and 0 for all other input combinations. Joining all products using a Boolean sum (OR) of all the product terms created, an algebraic representation of f is found. Using the example given in Table 3.1, and applying the rules above, the Equation (3.6) is obtained.

$$f = \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} + x_1 \cdot \overline{x_2} \cdot \overline{x_3} + x_1 \cdot x_2 \cdot \overline{x_3} \tag{3.6}$$

3.2.1 Literals

A literal is either a variable or the negation of a variable within a Boolean logic expression. For example, the expression represented by function $f = (x_1 + \overline{x_2}) \cdot (x_3 + \overline{x_1} \cdot x_4)$ has 5 literals and the variable set is $[x_1, x_2, x_3, x_4, x_5]$, being x_1 a positive literal and x_2 a negative literal.

3.2.2 Two Level Expressions

There are two ways to represent two level expressions. An expression called sum-of-products (SOP) is an expression that uses product terms joined by a sum. Another way is using expressions composed of sum terms joined by a product, being called product- of-sum (POS).

3.2.2.1 Minterms and Maxterms

For a Boolean function of n variables, a product term in which each of the n variables appears once (in its complemented or direct form) is called a minterm. Thus, a minterm is a logical expression of n variables that employs only the complement operator and the Boolean sum operator.

There are up to 2^n minterms for n variables since a variable in the minterm expression can be either in its complemented form or direct form.

Maxterms are similar to minterms. For a Boolean function of n variables, a sum term in which each of the n variables appears once (in its complemented or direct form) is called a maxterm.

For example, consider a function with 3 variables with input assignment $[x1, x2, x3]$. The indexes are the decimal representation of the binary value. Index 6 is the minterm $x_1 \cdot x_2 \cdot \overline{x_3}$ (maxterm $\overline{x_1} + \overline{x_2} + x_3$), the input assignment is $[1, 1, 0]$ and the minterm is denoted as m_6 (maxterm as M_6). Similarly, m_5 is $x_1 \cdot \overline{x_2} \cdot x_3$ (M_5 is $\overline{x_1} + x_2 + \overline{x_3}$) with input assignment $[1, 0, 1]$, and m_7 is $x_1 \cdot x_2 \cdot x_3$ (M_7 is $\overline{x_1} + \overline{x_2} + \overline{x_3}$) with input assignment $[1, 1, 1]$.

3.2.2.2 Implicants, Prime Implicants and Essential Prime Implicants

In Boolean logic, an implicant is a covering (sum terms or product terms) of one or more terms in a SOP (or maxterms in a POS) of a Boolean function. Implicants are also known as cubes. Considering a SOP, a product term p is an implicant of the Boolean function f if p implies f. The product term p implies f (and thus is an implicant of f) if f is equal one whenever p is equal one at the output. This concept can be extended to a POS.

A prime implicant pi of a function f is an implicant that cannot be covered by a more reduced (meaning with fewer literals) implicant. A prime implicant of f is a minimal implicant. The removal of any literal from pi results in a non-implicant for f. Essential prime implicants are prime implicants that cover an output of the function that no combination of other prime implicants can cover (WAGNER; REIS; RIBAS, 2006).

The process of removing literals from a term is called expanding the term. Expanding by one literal doubles the number of input combinations for which the term is true (in Boolean algebra). The sum of all prime implicants of a Boolean function is called the complete sum of that function.

For instance, in the function $f(x_1, x_2, x_3) = \overline{x_1} \cdot \overline{x_3} + x_1 \cdot x_3 + x_1 \cdot \overline{x_2}$, the Karnaugh map is shown in Figure 3.5.

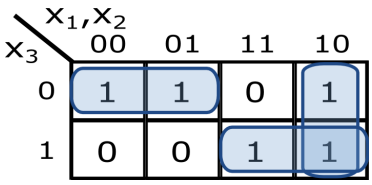


Figure 3.5 – Karnaugh map of function f.

An irredundant sum-of-products (ISOP) is a SOP where each product is a prime implicant and no product can be deleted or without changing the function. The irredundant product-of-sums (IPOS) is a POS where each sum is a prime no sum can be deleted without changing the function.

3.2.3 Factored Expressions

Factoring is the process of deriving a parenthesized algebraic equation, multilevel expressions, or factored form, representing a given logic function (BRAYTON, 1987).

An argument for factored forms is that they are a natural multilevel representation. A factored form is isomorphic to a tree structure, where each internal node is an AND or OR operator, each leaf is a literal, and the root node is the function output. This leads to a simple and relatively efficient multilevel implementation of the function of the output node. For instance, a function f can be expressed in a two level expression, represented by Equation (3.7). The Equation (3.7) can be factored in a more compact, parenthesized representation represented by Equation (3.8).

$$f = a \cdot c + b \cdot c + \overline{a} \cdot b \cdot d \tag{3.7}$$

$$f = (a + b) \cdot (c + \overline{a} \cdot d) \tag{3.8}$$

The logic tree of the two level expression and the factored expressions are shown in Figure 3.6 and in Figure 3.7, respectively. Note that the logic tree of the factored expression has

three levels of Boolean operations. The number of literals is also reduced, from 7 literals in the SOP expression to 5 literals in the factored expression.

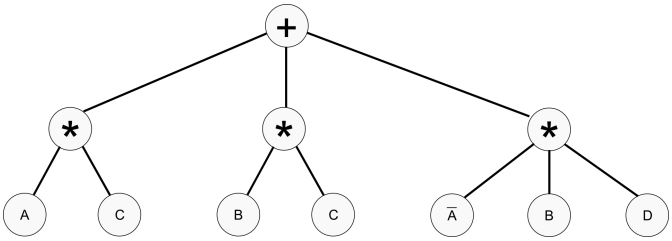


Figure 3.6 – L-cut of the two level expression representation of the expression $a \cdot c + b \cdot c + \overline{a} \cdot b \cdot d$.

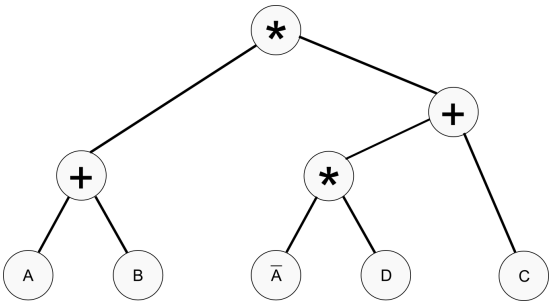


Figure 3.7 – Logic tree of the factored expression representation of the expression $(a + b) \cdot (\overline{a} \cdot d + c)$.

3.3 AIGs

An And-Inverter Graph (AIG) is a restricted type of a Directed Acyclic Graph (DAG), where each node has either 0 incoming edges - the *primary inputs* (PI) - or 2 incoming edges - the AND nodes. Each edge can be negated or not. Some nodes are marked as *primary outputs* (PO). An example of AIG is depicted in Figure 3.8.

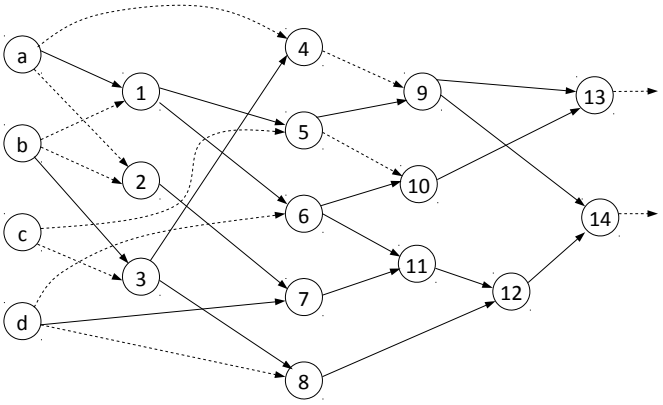


Figure 3.8 – And-inverter graph (AIG) representing a circuit (MACHADO et al., 2013).

3.3.1 K-cuts on AIGs

A cut of a node n in an AIG is a set of nodes c such that every path between a PI and n contains a node in c. A cut is said to be irredundant if no subset of the cut is also a cut. A K-cut of an AIG is an irredundant cut of up to k inputs. Let A and B to be two sets of cuts and let the auxiliary operation $\bowtie$ to be as described in Equation (3.9).

$$A \bowtie B \equiv \{a \cup b \mid a \in A, b \in B, \mid a \cup b \mid < k\} \tag{3.9}$$

Notice that the $\bowtie$ operation is commutative since the $\cup$ operation is also commutative. Also, $A \bowtie B$ is empty if either set is empty. Moreover, the $\bowtie$ operation can remove the redundant cuts by using signatures (MISHCHENKO; CHATTERJEE; BRAYTON, 2007).

3.3.2 KL-cuts on AIGs

To compute KL-cuts, it is necessary to compute initially the backward cuts or back-cuts (MARTINELLO et al., 2010). The idea is similar to calculate K-cuts, but it is performed backward, hence the name.

A backcut of a node n is a set of nodes c such every path between n and a PO contains a node in c. A backcut is irredundant if no subset is a backcut. An L-feasible backcut is an irredundant backcut containing l or lesser nodes.

Using a similar notation in Section 3.3.1, let define $\bowtie_{i=m}^{n} x_i \equiv x_m \bowtie \ldots \bowtie x_n$. This attribution can be made since $\bowtie$ operation is commutative.

Let $\Phi_l(n)$ to be the set of L-feasible backuts of n and let n_i to be the i-th node connected to its output. $\Phi_l(n)$ is defined as:

$$\Phi_L(n) = \begin{cases} \{n\} & n \text{ is a PO} \\ \{n\} \cup (\bowtie_i \Phi_L(n_i)) & otherwhise \end{cases} \tag{3.10}$$

The KL-cut defines a sub-graph $\mathcal{G}_k l$ of $\mathcal{G}$ which has no more than k inputs and no more than l outputs (MARTINELLO et al., 2010). The algorithm to compute KL-cuts is shown in Algorithm 1. The algorithm involves computing all K-cuts and all L-cuts (line 3-4) and for each l-cut, there is a combination of this L-cut for all K-cuts, and the primary inputs are used to create a KL-cut (line 6-8). If the KL-cut is valid (line 9), the KL-cut is added to the solutions. The algorithm returns all valid KL-cuts in the AIG. More details can be found in (MARTINELLO et

al., 2010).

Algorithm 1 KL-cutS Algorithm

```
1:  function COMPUTE_KLCUTS (k, l, aig)
2:      klcuts ← ∅
3:      kcuts ← COMPUTE_KCUTS (k, aig)
4:      lcuts ← COMPUTE_LCUTS (l, aig)
5:      for each lcut ∈ lcuts do
6:          p ← COMBINE_KCUTS (lcut, kcuts)
7:          for each pi ∈ p do
8:              klcut ← CREATE_KLCUT (pi, lcut)
9:              if CHECK_FIX (klcut) then klcuts ← klcuts ∪ klcut
10:     return klcuts
```

To illustrate the differences between K- and KL-cuts, we present a combinational circuit example composed of 8 inputs and 3 inputs, show in Figure 3.9. By enumerating the K-cuts with $k = 6$, the values given in Table 3.4 are obtained. Considering an unbounded L, the three KL-cuts found are shown with rectangles around the instances contained in each KL-cut (MACHADO et al., forthcoming).

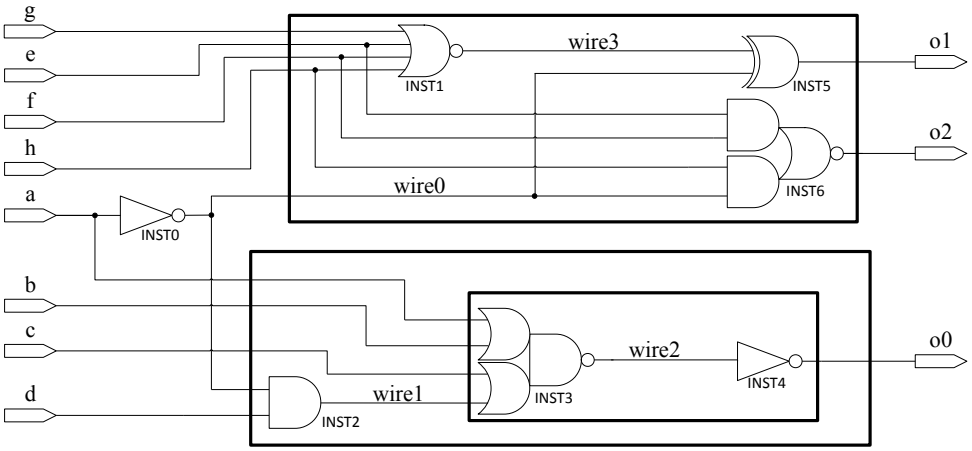


Figure 3.9 – Combinational circuit example to demonstrate the K-cuts and KL-cuts computation.

Table 3.4 – All K-cuts with $k = 6$ for all nodes of the combinational circuit example in Figure 3.9.

Node	K-cuts
a	{a}
b	{b}
c	{c}
d	{d}
e	{e}
f	{f}
g	{g}
h	{h}
wire0	{a}
wire1	{wire1}, {d, a}
wire2	{a, b, c, d}, {a, b, c, wire1}, {wire2}
wire3	{wire3}, {e, f, g, h}
o0	{a, b, c, d}, {a, b, c, wire1}, {wire2}
o1	{o1}, {a, wire3}, {a, e, f, g, h}
o2	{o2}, {a, e, f, h}
all nodes	{a, b, c, d}, {d, a}, {a, wire3}, {e, f, g, h}, {a, b, c, wire1}, {a, e, f, g, h}, {a, e, f, h}

3.4 Mapped circuits

A combinational mapped circuit C is a specific type of DAG with three types of nodes: the PI nodes, the *logic gate* nodes, and the PO nodes. If a node of C has no incoming edges and 1 or more outgoing edges, it is a PI. If a node of C has up to m incoming edges, where m is an integer value such that $m \geq 1$ and 1 or more outgoing edges, it is a logic gate node. If a node of C has 1 incoming edge and no outgoing edges, it is a PO. The main differences between the AIG and mapped circuit descriptions are:

1. the number of incoming edges, which are not limited by two in the mapped circuit;
2. the existence of inverters and buffers instead of simply negated or direct edges.

3.4.1 K-cuts on mapped circuits

Let $\Phi_K(n)$ to be the set of K-cuts of $n \in C$, and if n is a logic gate node, let $n_1, ..., n_g$ to be its inputs, where g is an integer value representing the number of inputs of n such that $m \geq g \geq 1$. By using the same operation $\bowtie$ described in Equation (3.9), $\Phi_K(n)$ is defined

recursively as described in Equation (3.11).

$$\Phi_K(n) = \begin{cases} \{n\}, & n \text{ is a PI} \\ \Phi_K(n_1), & \text{1 input cell} \\ \{n\} \cup \{\Phi_K(n_1) \bowtie \Phi_K(n_2) \bowtie ... \bowtie \Phi_K(n_g)\}, & \text{otherwise} \end{cases} \tag{3.11}$$

3.4.2 KL-cuts on mapped circuits

The KL-cuts in circuits are similar to the KL-cuts on top of AIGs (MACHADO et al., 2012), and it can be applied to improve a cost function of the cuts and then replace them in the original circuit. There are no KL-cuts formed by only one cell neither with $k = 1$ (e.g. inverter or buffer chains). One way to increase the shared logic for a given set of inputs, the l is defined as unbounded, not limiting the number of outputs and keeping track of all outputs that depend on the same set of variables.

To enumerate the KL-cuts of a circuit, it is necessary a k value (number of KL-cut inputs) and a mapped circuit. If the design has sequential elements, there is a need to treat these sequential elements as PIs and POs to the combinational logic. The algorithm starts by enumerating all K-cuts for all nodes of the circuit. The idea is similar to the presented in Algorithm 1. One of the advantages of a K-cut is finding redundancies in the logic, allowing further optimization. This is an important feature of K-cuts, in which the KL-cuts are based. Nevertheless, a K-cut generates only one output, i.e. a K-cut does not cover all outputs it affects. It is important to notice that KL-cuts provide a complete input-output interface for a sub-circuit substitution. Additionally, KL-cuts minimize the support of the Boolean functions inside the cut (MACHADO et al., 2012).

3.4.2.1 Polarity don't cares

After identifying a KL-cut instances, inputs, and outputs, a further search is performed on the inputs, identifying inverters and buffers. It is clear that the inverters and buffers used only to in the KL-cut can be inserted itself. However, if the inverters and buffers are also applied in other parts of the circuit, they cannot be inserted into the KL-cut, since it will generate a duplication of these cells. In this sense, the inverters that are not inserted can be used to generate a mapping flexibility: the *polarity don't cares* (MACHADO et al., 2013). A similar approach can

be performed on flip-flops that generate both polarities of a signal. For instance, a KL-cut found in a commercial benchmark is shown in Figure 3.10. Notice that the KL-cut has two *polarity don't cares* ($i9 = \overline{i0}$ and $i12 = \overline{i1}$).

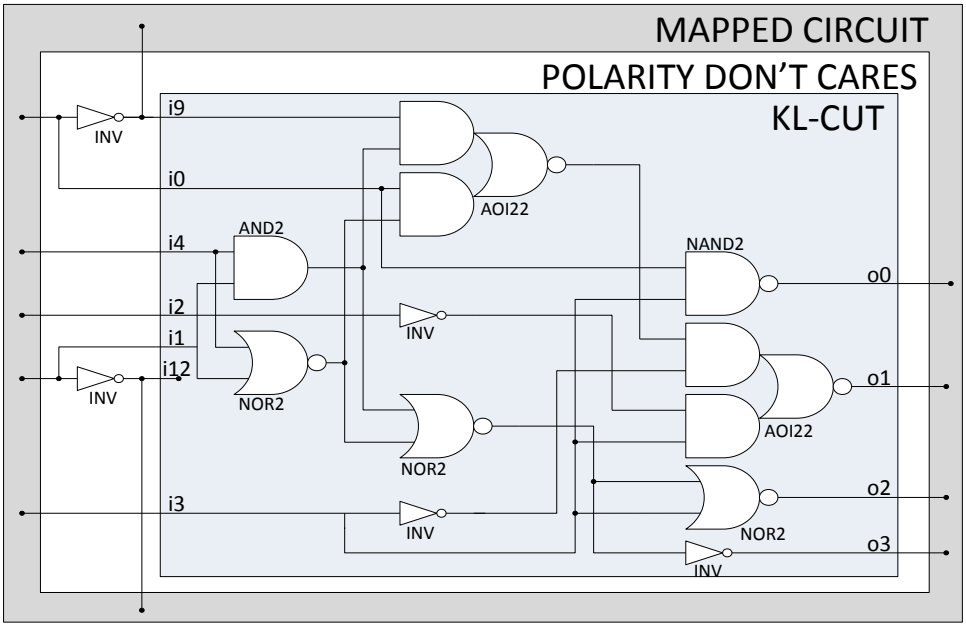


Figure 3.10 – Example of KL-cut found in a commercial benchmark with *polarity don't cares* assigned (MACHADO et al., 2013).

3.5 Functions and Sub-functions

As seen in the logic synthesis flow, the steps presented manipulate algebraic and Boolean representations to transform/manipulate the functions that represent the circuit. As the Boolean functions increase exponentially in size (i.e. the size of a truth table is 2^{2^n}, n being the number of variables or the support of the function), functions with a large number of variables can be cumbersome to manipulate. In this sense, subfunctions can have a significant role in algorithms that manipulate Boolean functions. Sub-functions are functions that represent small logic parts present in a target function.

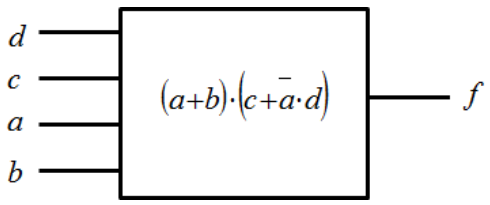


Figure 3.11 – Implementation of logic function f.

Table 3.5 – Truth table for a logic function with 4 variables.

m	a	b	c	d	f
m0	0	0	0	0	0
m1	0	0	0	1	0
m2	0	0	1	0	0
m3	0	0	0	1	0
m4	0	1	0	0	0
m5	0	1	1	1	1
m6	0	1	0	0	1
m7	0	1	0	1	1
m8	1	0	1	0	0
m9	1	0	1	1	0
m10	1	0	0	0	1
m11	1	0	0	1	1
m12	1	1	1	0	0
m13	1	1	1	1	0
m14	1	1	0	0	1
m15	1	1	0	1	1

For instance, in the example of Table 3.5, the truth table represents the function f. This function implements Equation (3.8) and is represented in Figure 3.11.

The function f can also be represented using two subfunctions, as illustrated in Figure 3.12, which the two functions are simpler than f since they have a support of 3 variables. The implementation is represented in Figure 3.13.

m	***a***	***c***	***d***	***X***
m0	0	0	0	0
m1	0	0	1	1
m2	0	1	0	1
m3	0	1	1	1
m4	1	0	0	0
m5	1	0	1	0
m6	1	1	0	1
m7	1	1	1	1

m	***a***	***b***	***X***	***f***
m0	0	0	0	0
m1	0	0	1	0
m2	0	1	0	0
m3	0	1	1	1
m4	1	0	0	0
m5	1	0	1	1
m6	1	1	0	0
m7	1	1	1	1

Figure 3.12 – Two truth tables representing subfunctions of the logic function f.

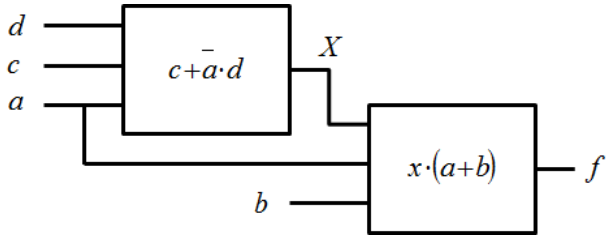


Figure 3.13 – Implementations of the two subfunctions of Figure 3.12.

3.5.1 Functional Decomposition

Functional decomposition is a method for combinational logic synthesis in which a Boolean function is decomposed into a set of subfunctions. The functional decomposition has been introduced by the pioneering works of Ashenhurst (ASHENHURST, 1957) and Curtis (CURTIS, 1962). The results of this technique are in the functional domain, meaning that it can produce non-trivial logic rewritings that are very suitable to overcome the structural bias (CHATTERJEE et al., 2006). Functional decomposition has been extensively used in FPGA mapping since it is easy to control the number of inputs at each sub-function (SASAO, 1993). When the functional decomposition can decompose a target function, only one decomposition is provided.

There are many related works on functional decomposition, such as disjoint support decomposition (DSD) (BERTACCO; DAMIANI, 1997; SASAO; MATSUURA, 1998; MINATO; MICHELI, 1998) and bidecomposition (YANG; CIESIELSKI, 2002; MISHCHENKO; STEINBACH; PERKOWSKI, 2001; CHOUDHURY; MOHANRAM, 2010).

3.5.1.1 Disjoint Support Decomposition

The disjoint support decomposition of a Boolean function $F(x_1, \ldots, x_n)$ consists in representing f by means of simpler component functions J and K, such that the inputs of J and K do not share any input variable, and $F = K(x_1, \ldots, x_{j-1}, J(x_j, \ldots, x_n))$. This DSD is shown in Figure 3.14. In general, a function has several disjoint support decompositions, which can be superimposed to obtain decompositions with finer granularity. Moreover, it is possible to search recursively for DSDs for functions J and K to produce even smaller components. At the limit, f can be represented as a tree of functions, with the inputs x_i being the leaves of the tree. In the example of Figure 3.6, a DSD algorithm is not able to decompose f since in a DSD

function each variable appears only once and in the logic function presented, the variable a appears twice.

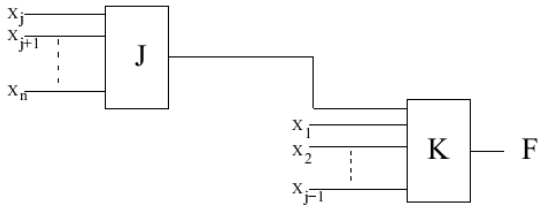


Figure 3.14 – A disjoint support decomposition for F. (BERTACCO; DAMIANI, 1997)

3.5.1.2 Bi-Decomposition

The bi-decomposition is a functional decomposition where a function is recursively decomposed into two smaller functions. Bi-decomposition algorithms generally apply multi-level *and, or, xor* decompositions. These algorithms rely on the ability to split the given logic function into two functions depending on fewer variables. One the most important steps in bi-decomposition algorithms are to determine a good variable partition since it affects both execution time and quality of results. The bi-decomposition can be classified in strong or weak bi-decomposition. These classifications are based on the support of the decomposed functions. The bi-decomposition is schematically represented in Figure 3.15, where the H represents the gate that connects the functions F and G. The original support S is divided in three parts, S_f, S_g and $S_i nt$. The variables in S_f are only presented in the function F, the variables in S_g are only presented in the function G and the variables in $S_i nt$ are presented in both. A weak bi-decomposition is characterized when S_f and S_g is empty. Otherwise, is considered strong.

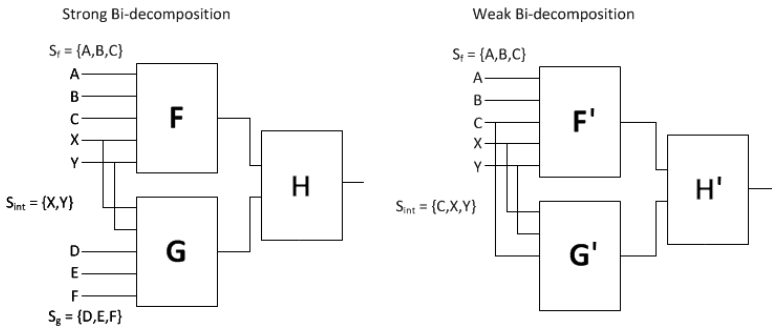


Figure 3.15 – Representation of a weak / strong bi-decomposition.

Considering the two decompositions presented, they have two critical drawbacks in this context. First of all, they are a top-down approach, which decomposes the original function into

smaller ones. Thus, the implementation cost of these functions is not necessarily known at the time of decomposition. Secondly, the functional decomposition depends on costly operations for one possible decomposition, relying on complex operations as count subfunctions extracted, test inversions, and so on.

3.5.2 Factoring Algorithms by Division

In mathematics, factorization or factoring is the decomposition of an object (for instance a polynomial) into a product of other objects, or factors, which when multiplied together give the original. Algorithms that factorize Boolean expressions can be divided into two groups: those who use algebraic techniques and those who use Boolean techniques.

The algebraic factoring or weak division (BRAYTON, 1987) is a factorization method that treats a Boolean expression as a polynomial of real numbers. The basic concept is that given the functions f and p, find functions q and r such that $f = p \cdot q + r$, if such q and r exist. This operation is called the division by p generating quotient q and the remainder r. The function p is called a divisor of f if$r$ is not null, and a factor if r is null. The algebraic model does not consider monotone laws as idempotence, absorption, annihilation, identity and distributive laws and nonmonotone laws and complementation, double negation and De Morgan laws.

The input and divisor needs to be represented in two-level sum-of-products (SOP) forms. A sum of products (SOP) is cube-free if no cubes (except '1') are able to divide another SOP. For instance, $x_1 \cdot x_2 + x_3$ is cube-free, whereas $x_1 \cdot x_2 + x_1 \cdot x_3$ is not cube-free, since x_1 is common to both cubes. A kernel is a cube-free quotient obtained by dividing a Boolean expression X by a single cube c. This single cube c is called co-kernel of X. In the previous example, $x_2 + x_3$ is a kernel and x_1 is a co-kernel.

For a given division operation, the resulting q and r may depend on upon the particular representation of f and p. Moreover, for any logic function, there are many factors and divisors. This fact poses a problem in choosing a good factor and divisor. If the domain is restricted to a particular subset of expressions, then the division operation is unique and much easier to carry out. For instance, the Equation (3.12) is an algebraic product.

$$f = (x_1 \cdot x_2 + x_3) \cdot (x_4 + x_5) \tag{3.12}$$

Unlike algebraic factoring, Boolean factoring exploits Boolean identities and Boolean properties to perform factoring (e.g. the annihilation property: $a + 1 = 1$), allowing products

with variables in common. For instance, Equation (3.13) is an example of Boolean factorization.

$$g = (x_1 \cdot x_2 + x_3) \cdot (\overline{x_1} + x_2 \cdot x_5) \tag{3.13}$$

Note that Equation (3.12) and Equation (3.13) are different and Equation (3.13) allows products that are not observed in algebraic factoring. If Equation (3.12) is expanded, the Equation (3.14) is found:

$$g = x_1 \cdot x_2 \cdot \overline{x_1} + x_1 \cdot x_2 \cdot x_2 \cdot x_5 + x_3 \cdot \overline{x_1} + x_3 \cdot x_2 \cdot x_5 \tag{3.14}$$

The first product can be eliminated by the complementation law ($x \cdot \overline{x} = 0$) and the second product can be simplified by idempotence law ($x \cdot x = x$). The Equation (3.15) is the Equation (3.12), considering the discussed simplifications.

$$g = x_1 \cdot x_2 \cdot x_5 + x_3 \cdot \overline{x_1} + x_3 \cdot x_2 \cdot x_5 \tag{3.15}$$

Algebraic factoring is very fast, but the quality of results is far from optimal. The Boolean factoring usually achieves better results, but they can be very time and memory consuming. Algebraic algorithms treat the Boolean expression as a polynomial, which reduces the execution time, but the final result is strongly tied to the starting expression (i.e. the initial expression that the algorithm uses as a basis to factorize). In the example presented in Figure 3.6, algebraic algorithms are not able to find the minimal implementation (i.e. the result will be $x_1 \cdot x_2 + x_1 \cdot x_3$ whereas Boolean factoring algorithms are capable of finding the minimal implementation.

3.5.3 Functional Composition

The main idea functional composition is the opposite of the functional decomposition. Instead of starting from a function with large support and breaking this function in subfunctions with smaller support, functional composition starts from basic subfunctions (e.g. literals) and composes them in more complex subfunctions until the target function is achieved. Figure 3.16 shows that functional composition is able to find the minimal solution for the logic function presented in Figure 3.6. Each arrow represents a composition, and two simpler functions are combined to compose a more complex function. For instance, a and b functions are combined by an OR operation to compose the $a + b$ logic function.

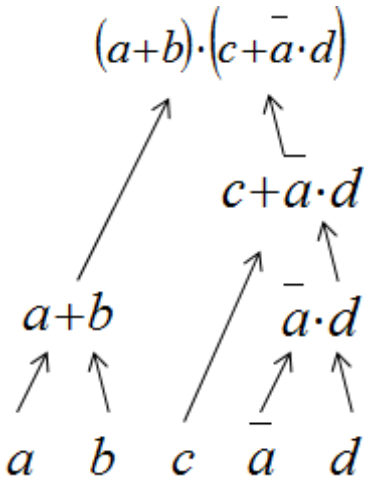


Figure 3.16 – Functional composition for the logic function of Figure 3.7

It is easy to see that the functional composition should not be restricted to AND, OR, XOR operators. The composition operations can be non-trivial ones, as majority functions, material implication functions, multiplexer functions, and others. These operations represent the logic implemented in basic gates of a set of emerging technologies (LENT et al., 1993; FAHMY; KIEHL, 1999; AVERIN; LIKHAREV, 1986; CHUA, 1971; PRODROMAKIS; TOUMAZOU; CHUA, 2012; STRUKOV et al., 2008; ZHU et al., 2013; PERSHIN; VENTRA, 2012; FRIEDMAN et al., 2012a; FRIEDMAN et al., 2012b).

In this sense, it is important to investigate the characteristics of functional composition since there are few dedicated algorithms able to synthesize logic using gates different from basic CMOS ones (AND, OR, XOR). The functional composition will be explored in the next chapter.

4 FUNCTIONAL COMPOSITION OVERVIEW

In this chapter, we present a more detailed explanation about the functional composition (FC) and its five principles. These principles allow a systematic approach and turn easy different implementations of this technique (MARTINS; RIBAS; REIS, 2012).

4.1 Principles

The principles used in FC include the use of bonded-pair representation, the use of initial functions set, the association between simple functions to create more complex functions, the control of costs achieved by using a partial order that enables dynamic programming, and the restriction of allowed functions to reduce execution time/memory consumption. These general principles are discussed in the following subsections.

4.1.1 Bonded-Pair Representation

FC uses bonded-pairs to represent logic functions. The bonded-pair is a data structure that contains a functional and a structural representation of a Boolean function. The functional and structural representation must be logically equivalent. The functional representation needs to be a canonical representation, as a truth table or a reduced ordered binary decision diagram (ROBDD) structure. The structural representation is related to the implementation desired and is used to control costs in the final implementation. Since the functional part is canonical, there is no necessity to have a canonical implementation, as costs may vary. We will denote a bonded-pair by the following notation: $\langle F, S \rangle$, where F represents the functional part and S the structural part. Figure 4.2 is illustrated an example of a bonded-pair representation with structural part implemented as a logical expression and the functional part as a truth table represented as an integer, considering the most significant bit the leftmost.

Functional Part	Structural Part
1101_2	$\overline{a} + b$

Figure 4.1 – Bonded-pair representation using a truth table and a expression as the functional and structural part, respectively.

4.1.2 Bonded-Pair Association

The bonded-pair association is a logic operation (e.g. logic XOR) being applied independently to the functional and the structural parts. By applying the same operation in functional and structural representations, the correspondence between the representations is still valid after such operation. The conversion of functional representation into a structural representation, and vice-versa may be challenging and inefficient. The main advantage of the bonded-pair association is the operations occurring in the functional and structural domain in parallel, avoiding conversions. The bonded-pair representation is used to maintain the control over the structural representation, avoiding the inefficient structures and a lack of control of converting one type into another. The bonded-pair association is not limited to binary operations. Actually, it can be performed with n-ary operators. Figure 4.3 presents the association of bonded-pairs. The bonded-pair $\langle F_3, S_3 \rangle$ is obtained from bonded-pairs $\langle F_1, S_1 \rangle$ and $\langle F_2, S_2 \rangle$. The computation of the functional part ($F_3 = F_1 + F_2$) is independent of the computation of the structural part ($S_3 = S_1 + S_2$).

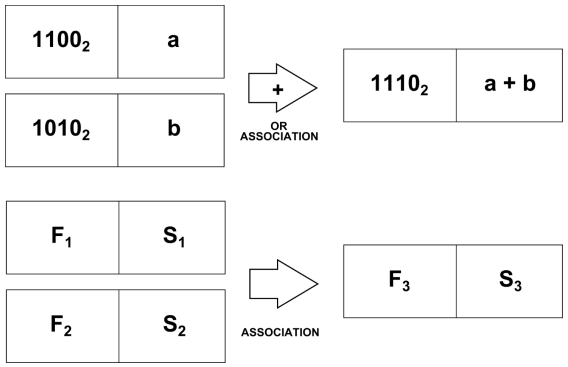


Figure 4.2 – Bonded-pair association. Notice that the operations occur independently in functional and structural part.

4.1.3 Initial Bonded-Pairs

As seen previously, a way to computes new bonded-pairs is associating known bonded-pairs. As a consequence, a set of initial bonded-pairs is necessary before starting the algorithm. The set of initial bonded-pairs have two main characteristics: (1) the initial bonded-pairs are the initial input of any algorithm based on FC; (2) the initial bonded-pair must have known implementations costs (preferable minimum costs) for each bonded-pair, allowing the computation of the cost for derived functions. For instance, in Figure 4.3 is illustrated a possible set of initial bonded-pair with two variables, using the bonded-pair representation shown in Figure 4.1.

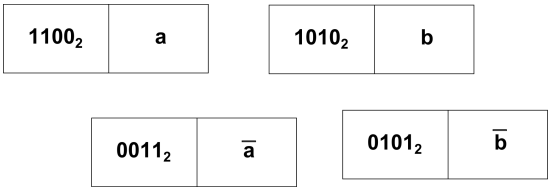


Figure 4.3 – Initial bonded-pairs, considering a 2-variable problem.

4.1.4 Dynamic Programming and Partial Order

The key concept of dynamic programming is solving a problem in which its optimal solution is obtained by combining optimal sub-solutions. This concept can be applied to problems that have optimal substructure. It starts by solving sub-problems and then combining the sub-problem solutions to obtain a complete solution. In this sense, it is necessary that the problem has an optimal sub-structure. One example of a problem with an optimal sub-structure is the shortest path between two nodes in an unweighted graph. But not all problems have optimal substructure, as the longest simple path in an unweighted graph. In this sense, the bonded-pair representation allows applying the memoization concept. A memoized algorithm maintains an entry in a table for the solution for each subproblem. The memoization occurs because the bonded-pair representation stores the implementation of the functions. The intermediate bonded-pairs functional part can be thought as the subproblems and the structural part as the subsolutions. In functional composition, dynamic programming is used associated with the concept of partial ordering. The partial ordering classifies elements according to some cost. This is done to ensure that implementations (the structural elements in the bonded-pairs) with minimum costs are used for the sub-problems. Different costs can be used depending on the target(s) to be minimized. Using the concept of partial order, intermediate solutions of subproblems are classified into 'buckets' that sort them in an increasing order of costs of the structural element of the bonded-pair representation. This concept is illustrated in Figure 4.4, where the number of literals was chosen as the partial order criteria.

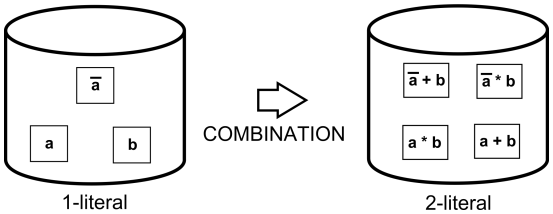


Figure 4.4 – A partial order considering the number of literals.

4.1.5 Allowed subfunctions

The large number of subfunctions, created by exhaustive combination, can jeopardize the FC approach. However, many optimizations can be done to make FC approach feasible and more efficient. One of these optimizations is the use of the allowed functions. For performance optimization, a hash table of allowed functions can be pre-computed before starting the algorithm. Functions that are not present in the allowed functions table are discarded during the processing. The use of the allowed functions hash table helps to control the execution time and memory use of the algorithms. FC may (in some cases) achieve a better result by having more allowed functions than with a reduced set of these. For other cases, solutions can be guaranteed optimal even with a very limited set of allowed functions. Several effort levels can be implemented for the trade-off memory/execution time versus quality. These effort levels can vary from a limited set of functions to an exhaustive effort including all possible functions. An example of allowed functions is shown in Figure 4.5, based on the example shown in Figure 4.4. A heuristic algorithm discarded the function $\overline{a} \cdot b$, reducing the amount of functions inserted in the bucket.

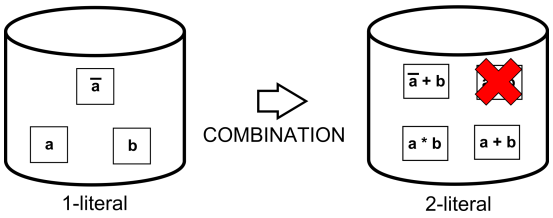


Figure 4.5 – An heuristic can be applied to reduce the number of bonded-pairs in each bucket.

4.2 Relationship to Dynamic Programming

There are several problems in the logic synthesis that can take advantage of FC, but as seen in Section 4.1.4, the dynamic programming takes an important role. Therefore, some steps are necessary before.

The first step is to characterize the structure of an optimal solution. If the problem does not have an optimal solution, composed by optimal subsolutions, the dynamic programming will fail in its goal. The second step is recursively defining the value of the optimal solution. This means that the problem needs to be broken in subproblems, each one implemented with an optimal subsolution until reaching the initial input (i.e. the initial bonded-pairs). This is the reason why the initial bonded-pairs must be implemented with a minimal cost structure.

The third step is to compute the value of an optimal solution, typically in a bottom-up fashion. This step represents the bonded-pair association, where the functional (subproblem) and structural (subsolution) parts are combined. This solution will have an optimal implementation since the subsolutions were also optimal. The last step is to construct an optimal solution from the computed information. This step is important if we want to know the implementation of the solution. If we do not store the computed information (implementation), we will only have the cost of implementation, provided by the partial order.

As observed in the "allowed functions" principle, some functions can be discarded in the process, reducing the solution space, but as a consequence, it may provide solutions that are unboundedly below the optimal. This concept is called bounded dynamic programming.

4.2.1 Bounded Dynamic Programming

The main limitation of optimal approaches using dynamic programming is that the number of solutions stored is sometimes prohibitive, resulting in huge memory requirements for all but the simplest problems. To mitigate these requirements, some restrictions are applied to ensure that these requirements (e.g. memory, execution time) are attended. One of the examples of bounded dynamic programming is the beam search algorithm. The beam search is a heuristic search algorithm that traverses a graph by expanding the most promising node in a limited set. Beam search is an optimization of best-first search that reduces its memory requirements. Best-first search is a graph search which orders all partial solutions (states) according to some heuristic which attempts to predict how close a partial solution is to a complete solution (goal state). However, in beam search, only a predetermined number of best partial solutions are kept as candidates.

4.3 Relationship to Genetic Programming

In genetic programming (GP) we evolve a population of programs. That is, GP randomly transforms populations of programs into new, hopefully, better populations of programs. Due to GP randomness, it can never guarantee results. However, this can lead ways to escape minimum locals that algorithms can be trapped into. The primary genetic operations that are used to create new programs from the existing ones are crossover and mutation.

Crossover is a creation of a child program by combining randomly chosen parts from two

selected parent programs. Mutation is a creation of a new child program by randomly altering a chosen part of a selected parent program.

FC can be though as GP since there are exhaustive combinations from initial bonded-pairs (crossover) that occur randomly (respecting the partial order concept). The mutation concept is not applied since this would break the equivalence between the functional and structural parts in a bonded-pair.

4.3.1 Costs in Functional Composition

The structural part in the bonded-pair most of the time has not a canonical form. To compare different structures, it is necessary to adopt a figure of merit or cost. Cost is a quantity used to characterize the structure, about its alternatives. A simple example of structural cost can be the total area from the gates in a circuit. Also, each structure can contain multiple costs, as the logic depth or the number of literals in a factored form. Usually, the minimization of multiple costs can be conflicting. A classical problem is the area *vs* timing *vs* power optimization in circuits, being very difficult (sometimes impossible) to optimize all 3 at the same time.

There are several strategies to optimize multiple implementation costs using FC. One of them is the ranking strategy when the structures are selected optimizing the first cost (highest ranking). When there is a tie, the second cost is used as tie-breaker, and so on. Another strategy is using a weighted arithmetic mean. Each cost has attributed a weight, and the structure with the smaller mean is selected. These strategies can provide very different results, depending on the structure being optimized.

4.4 General Algorithm of Functional Composition

Algorithm 2 presents a generic version for the functional composition, which will be used as the basis for all applications that will be introduced in the next chapters. The $target$ on the FC_SYNTHESIZE method can represent a function, a list of functions or the universe of Boolean functions up to n inputs. The $\langle initial, idx \rangle$ list describe a tuple with the initial bonded-pairs indicated by its initial structural cost. These initial bonded-pairs can be computed internally or externally of the method, depending on the application. The $allowed$ represent a set of allowed functions, which also can be computed internally or externally. As the allowed functions are optional, this set can be empty, indicating to the algorithm that there are no specific

functions to be allowed or not (i.e. allowing all functions). The algorithm starts allocating all initial bonded-pairs in their correct buckets (line 3-4). The NEXT method (line 5) will get the index of the first bucket to be composed. The main loop (line 6-8) represents the partial order and dynamic programming principles using bonded-pair associations, which will synthesize intermediate functions until the target is achieved. The COMBINE_METHODS method (line 7) will use the previous buckets and the allowed functions set to generate the next bucket using bonded-pair associations. After the target is achieved, the solution (or solutions) are returned (line 9).

Algorithm 2 FC-SYNTHESIS Algorithm

```
1: function FC_GENERIC (target, ⟨initial, idx⟩, allowed)
2:     B ← ∅
3:     for each init ∈ initial with index ∈ idx do
4:         B[idx] ← init
5:     i ← NEXT (⟨initial, idx⟩)
6:     while target is not achieved do
7:         B[i] ← COMBINE_BUCKETS(B, i, allowed)
8:         i ← i + 1
9:     return solution
```

4.5 Related Work

The use of a bottom-up approach for logic synthesis was already used by some authors, including (JÓZWIAK; BIEGAŃSKI, 2008) and (HLAVIČKA; FIŠER, 2001).

The FC is different from the work of (JÓZWIAK; BIEGAŃSKI, 2008), because Joswiak does bottom-up synthesis by using information theory (JÓŹWIAK, 1999), An information-driven circuit synthesis approach relies on the analysis of the information flow structure and relationships in the function to be implemented, as well as, in the circuit under construction, and usage of the results of this analysis to control the circuit construction. Based on information theory, the algorithm classifies subfunctions which will better contribute to cover a given function.

This process is directly performed into the primitives of a given implementation technology (e.g. gates of a given technology library), while FC performs it by an extensive combination of bonded-pairs, manipulating the functional and structural parts. Notice that using information theory implies computing the information with a method that has to visit every minterm of a function individually. So, information theory computation is more expensive than the bitwise

operations with integers representing truth tables, which can be used in FC. The key enabler of FC presented herein is the concept of bonded-pairs which was explained in this chapter, as bonded-pair association guarantees that a fast computation of functions from subfunctions, which enables to exploit more implementations.

The FC also differs from the work of (HLAVIČKA; FIŠER, 2001), because Hlavicka relies on the bottom-up approach to compute only an incomplete set of prime implicants, performing two-level minimization. The algorithm starts from a functional description and has three phases. The three phases are coverage-directed search (generation of implicants); implicant expansion (generation of prime implicants) and solution of the covering problem. The coverage-directed search consists of a directed search for the most suitable literals that should be added to some previously constructed term to convert it into an implicant of the given function. Thus instead of increasing the dimension of an implicant starting from a 1-minterm (or any other 1-term given in the function definition), we reduce the n-dimensional cube by adding literals to its term, until it becomes an implicant of the given function. These implicants generated during this phase are not necessarily prime implicants.

In the implicant expansion, the cubes are expanded, which means by removing literals (variables) from their terms. When no literal can be deleted from the term anymore, a prime implicant is generated. Having found a sufficient set of prime implicants, the covering problem is solved. The heuristics used correspond to the method suggested in (RUDELL; SANGIOVANNI-VINCENTELLI, 1989; COUDERT, 1994). The algorithm is capable of dealing with functions with several hundreds of input variables, competing with ESPRESSO. In FC approach, the complete synthesis process is based on dynamic programming by the association of bonded-pairs. The FC is a general method that can be applied to several applications instead of only prime implicants computation.

5 CMOS APPLICATIONS USING FUNCTIONAL COMPOSITION

In this chapter, it will be presented two applications using FC for CMOS technology. The first application presents a multi-output factorization algorithm which is applied in a resynthesis framework (MACHADO et al., 2012; MACHADO et al., 2013; MACHADO et al., forthcoming). The second generates approximate functions that can be used to synthesize ATMR circuits (GOMES et al., 2014; GOMES et al., 2015a; GOMES et al., 2015b). Both applications are based on a Boolean factorization algorithm, which uses functional composition (MARTINS et al., 2010). It is important to notice that even these applications were developed having the CMOS as main technology, both concepts (multi-output factorization and approximate function generation) can be applied to any technology with few modifications.

5.1 Multi-output Factorization applied in Circuit Resynthesis

Factoring is an important procedure in logic synthesis tools. It consists in converting a logic function into a logically equivalent parenthesized expression or factored form with the goal of reducing the literal count. The factoring algorithms are usually divided into two-level synthesis, used mainly in Programmable Logic Array (PLA), and multilevel synthesis. In the two-level synthesis, there are tools as ESPRESSO (BRAYTON, 1987) that can find a minimum or near-minimum sum of products form for a logic function.

Multilevel synthesis is still in research, being the main implementation strategy used in the industry today. This section presents an algorithm that optimizes multi-output functions using FC. The performed Boolean optimizations allow a more extensive use of complex gates in the technology mapping step, reducing the area. To demonstrate the efficiency and applicability of the proposed algorithm, it is necessary a collection of other algorithms and techniques. Also, since FC performs a considerable number of Boolean operations, it may not be able to address large circuits. Thus, it is necessary a method that extracts Boolean functions from circuits (K-cuts or KL-cuts) (MACHADO et al., 2012) to divide the circuit into smaller parts. These smaller parts can be now synthesized using FC. Thus, the circuit can be traversed (through cuts) and optimized.

In this section, We start presenting a baseline single-output factoring algorithm. To reduce the execution time, we apply heuristics in the baseline algorithm. The single-output heuristic factoring algorithm is modified to handle multi-output functions, using the Boolean information to find logic sharing between the outputs. To validate this technique, this algorithm is applied to an iterative resynthesis flow, considering timing constraints. The resynthesis flow considers the use of KL-cuts which minimize the support of the Boolean functions (through dominance rules). The resynthesis flow also adopts a conservative approach, allowing only cut substitutions that reduce area and do not negatively impact the timing constraints.

5.1.1 Single-output Factorization Algorithm

This subsection is a basic review the factoring algorithm presented in (MARTINS et al., 2010) for the elaboration of the multi-output algorithm, which is based on the former. First, we will present a baseline algorithm using FC for factorization, which we will insert a heuristic to reduce the search space.

5.1.1.1 Baseline Factoring Algorithm

The baseline factoring algorithm computes new functions from simpler expressions (i.e., with fewer literals) computed in prior steps, to find the target function optimizing its number of literals. The starting point is the set of known sub-functions represented by single literals. The computation of new functions can be as follows. Let L, M and N be positive integer numbers with the following relations: $L \geq M$ and $N = L + M$. The procedure combines an L-literal function with an M-literal function, creating an N-literal function by using logic operations. An N-literal bucket is a set of N-literal functions. The operations among buckets combine all functions in an L-literal bucket against all functions in an M-literal bucket, generating an N-literal bucket ($N > 1$). The initial functions with 1-literal are inserted in the bucket with $N = 1$. Thus, the generation of the N-literal bucket can be expressed in Equation (5.1).

$$\bigcup_{i=1}^{\left|\frac{N}{2}\right|} ((B_i \cdot B_{N-1}) \cup (B_i + B_{N-1}))\,|N \geq 2 \tag{5.1}$$

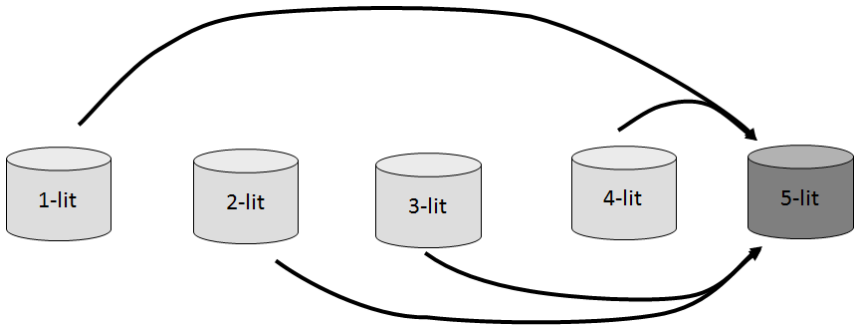


Figure 5.1 – Generation of functions contained in the 5-literal bucket (MARTINS et al., 2010).

In Equation (5.1), B_N represents the bucket and with index N. For example, in the Figure 5.1, the bucket 5 is formed by bonded-pair associations of elements in bucket 1 and bucket 4, and also by the association of elements in bucket 2 and bucket 3.

An important structure is the “already looked set”. The “already looked set” stores the functions already introduced. These functions have been produced with fewer or equal number of literals and do not need to be introduced twice. This process speeds up the execution time, decreasing the memory use.

The factorization process is iterative and stops when the target function is found. This function generation technique makes the algorithm find the optimal result in number of literals by construction.

Theorem 5.1.1 *The combination of the buckets using Equation* (5.1) *will result in a minimum literal expression, considering FC principles.*

Proof The 1-literal bucket contains all variables in both polarities, i.e., the positive and negative literals. Functions expressed as 1-literal forms are minimal since it is not possible to represent these functions with less than one literal. The 2-literal bucket contains all functions generated by combining functions of the 1-literal bucket. The functions in the 2-literal bucket have exactly 2 literals. Since neither constant functions nor functions are already present in buckets with smaller indexes are added (i.e. allocated in the memory), the newly added functions are known to be in the optimal form (in number of literals). By induction, the n-bucket is formed by functions with optimal form, generated using Equation (5.1). When the target function is found for the first time in the n-bucket, the minimum literal form is guaranteed to have n literals. ■

From a dynamic programming point-of-view, the algorithm has optimal substructure, as an optimal factored form is always a product or a sum of optimal factored forms. This process is iterative, stopping when the target function is found. However, the number of functions grows exponentially, as there are Boolean expressions of n inputs. If the functions in each bucket are not pruned, the algorithm becomes unfeasible in memory and computational time. In this sense, a heuristic approach (MARTINS et al., 2010) will be presented. The heuristic approach allows a limited set of functions because the functions that are not present in the allowed functions set are discarded, decreasing memory use and execution time, but in some cases losing the optimality.

5.1.1.2 Heuristic Factoring Algorithm

In the heuristic algorithm (FC-FACTOR-HEUR) (MARTINS et al., 2010), we consider only three types of functions: the smaller, larger and not comparable functions. The allowed functions in FC-FACTOR-HEUR are a set of functions derived from the cofactors of the target function. The initial step extracts all cofactors of the target function to associate them in the next step. Not comparable cofactors are associated using AND/OR operations to generate new not comparable/smaller/larger functions that are stored in allowed functions set. The second association is among not comparable and smaller cofactors using OR operation, only storing resulting larger functions. Similarly, there is the association among not comparable and larger cofactors using AND operation, only storing resulting smaller functions. All these functions (original cofactors and functions generated) are inserted in the “allowed functions set”. The set comprising all cofactors and its associations of the target function is a very good set of functions to compose the allowed functions set. The idea behind this concept is that it is possible to

Table 5.1 – AND/OR/XOR operations considering function order.

#	AND	#	OR	#	XOR
(1)	$SM \cdot SM$	(11)	$SM + SM$	(21)	$SM \oplus SM$
(2)	$SM \cdot LG$	(12)	$SM + LG$	(22)	$SM \oplus LG$
(3)	$SM \cdot NC$	(13)	$SM + NC$	(23)	$SM \oplus NC$
(4)	$SM \cdot DJ$	(14)	$SM + DJ$	(24)	$SM \oplus DJ$
(5)	$LG \cdot LG$	(15)	$LG + LG$	(25)	$LG \oplus LG$
(6)	$LG \cdot NC$	(16)	$LG + NC$	(26)	$LG \oplus NC$
(7)	$LG \cdot DJ$	(17)	$LG + DJ$	(27)	$LG \oplus DJ$
(8)	$NC \cdot NC$	(18)	$NC + NC$	(28)	$NC \oplus NC$
(9)	$NC \cdot DJ$	(19)	$NC + DJ$	(29)	$NC \oplus DJ$
(10)	$DJ \cdot DJ$	(20)	$DJ + DJ$	(30)	$DJ \oplus DJ$

obtain good sub-expressions of the formula, by setting variables to zero and one in an optimized factored form.

Algorithm 3 shows the pseudo code for the FC-FACTOR-HEUR. The algorithm has as inputs the target function (f) and the initial functions in their right polarities ($initial$). The first step is to test if the function is a constant (line 2). In this case, the solution is trivial. Also, if the solution is a variable, the FACTORIZE method will return the solution (line 5). Else, the algorithm starts computing the cofactors (line 6) and combining them (line 7), to compute the allowed functions. The buckets now will be combined (line 8-11), until the solution is found. The COMBINE_COFACTORS method separates the cofactors in smaller, larger and not comparable, (line 15-18) and implement the combinations (5), (8), (11) and (18), listed on Table 5.1, as the main combinations. The results of combinations (6) and (13) are conditional, to get functions that will contribute to the solution and do not increase the search space significantly. Each combination and its results is discussed in details in (MARTINS et al., 2012). The presented combinations guarantee that the algorithms always find a solution because in the worst case the presented combinations will generate a factored form based on the Shannon expansions observed in Equation (3.2), Equation (3.3) and Equation (3.4)

Algorithm 3 FC-FACTOR-HEUR Algorithm

```
1: function FACTORIZE (f, initial, useXOR)
2:     if IS_CONSTANT_FUNCTION (f) then return constant_value
3:     B ← ∅
4:     B[1] ← initial
5:     if CONTAINS (B[1], f) then return variable
6:     cofactors ← COMPUTE_COFACTORS (f)
7:     allowed_functions ← COMBINE_COFACTORS (cofactors, useXOR)
8:     i ← 2
9:     while f is not found do
10:        B[i] ← COMBINE_BUCKETS(B, i, useXOR)
11:        i ← i + 1
12:    return solution
13:
14: function COMBINE_COFACTORS (cofactors, useXOR)
15:    sm ← GET_SMALLER (cofactors)
16:    lg ← GET_LARGER (cofactors)
17:    nc ← GET_NOT_COMPARABLE (cofactors)
18:    allowed_functions ← ∅
19:    for each cofactor c1, c2 ∈ sm do
20:        allowed_functions ← OR (c1, c2)
21:    for each cofactor c1, c2 ∈ lg do
22:        allowed_functions ← AND (c1, c2)
23:    for each cofactor c1, c2 ∈ nc do
24:        allowed_functions ← AND (c1, c2)
25:        allowed_functions ← OR (c1, c2)
26:    for each cofactor c1 ∈ sm do
27:        for each cofactor c2 ∈ nc do
28:            f ← OR (c1, c2)
29:            if ORDER (f) = LARGER then allowed_functions ← f
30:    for each cofactor c1 ∈ lg do
31:        for each cofactor c2 ∈ nc do
32:            f ← AND (c1, c2)
33:            if ORDER (f) = SMALLER then allowed_functions ← f
34:    if useXOR then
35:        for each cofactor c1 ∈ nc do
36:            for each cofactor c2 ∈ nc do
37:                allowed_functions ← XOR (c1, c2)
38:    return allowed_functions
39:
40: function COMBINE_BUCKETS (B, i, useXOR)
41:    S ← ∅
42:    for k ← 1, (i/2) do
43:        l ← i − k
44:        S ← S ∪ COMBINE (B[k], B[l], useXOR)
45:    return S
```

5.1.1.3 XOR Support for FC-FACTOR-HEUR

The factoring with XOR is based on the FC-FACTOR-HEUR algorithm, appending the XOR operation between functions. Thus, the generation of the N-literal bucket can be now expressed as following:

$$\bigcup_{i=1}^{\left|\frac{N}{2}\right|} \left(\left(B_i \cdot B_{N-1}\right) \cup \left(B_i + B_{N-1}\right) \left(B_i \oplus B_{N-1}\right) \right) | N \geq 2 \tag{5.2}$$

The algorithm remains almost the same, except for the addition of combination (28) in the COMBINE_FUNCTIONS method. It is verified empirically that the XOR operation is only useful when there are at least 2 binate variables. In this case, the XOR use can be efficiently exploited if there are at least 2 binate variables. Otherwise, by using the XOR operator, only functions that not contribute to the solution are generated. This is explained by the fact that the XOR operator is binate since it carries both polarities of its input variables With the XOR operation. If the use of XOR is allowed, The COMBINE_COFACTORS method will perform the operation (28) in Table 5.1. Since the XOR operator has a physical implementation cost higher than the AND/OR operators, the COMBINE_FUNCTIONS method needs to consider different costs for AND/OR and XOR operations to choose the best implementation option.

5.1.2 Multi-output Factorization Algorithm

A heuristic multi-output factorization algorithm is proposed to synthesize multi-output circuits present in KL-cuts. We implemented an improved version of the algorithm presented in Section 5.1.1, considering the multi-output complexity. Let's consider the following example, already presented in the *polarity don't cares* subsection. Equation (5.3) presents the factored forms.

$$\begin{aligned} o0 &= \overline{i0} + \overline{i3} \\ o1 &= (\overline{i3} \cdot (\overline{i0} \cdot i1 \cdot i4 + \overline{i1} \cdot \overline{i4} \cdot i0)) + i2 \cdot i3 \\ o2 &= \overline{i3} \cdot (\overline{i1} \cdot \overline{i4}) + (i1 \cdot i4) \\ o3 &= \overline{i1} \cdot \overline{i4} + i1 \cdot i4 \end{aligned} \tag{5.3}$$

One interesting observation is multiple appearance of logic between the outputs. For

instance, $(i1 \cdot i4)$ and $(\overline{i1} \cdot \overline{i4})$ appears in $o1$, $o2$, and $o3$. Moreover, all the implemented logic in $o3$ appears in $o2$. Also, there is a possibility that $o1$ can be expressed using the Boolean information in $o2$. In this sense, one strategy could be identify the variable set (support) of each output function and considering that a output o_m has the possibility of be implemented using another output o_n if and only if $S(o_n) \subset S(o_m)$, where $S(f)$ denotes the support of the Boolean function f. With the presented strategy, we can derive the dependence $D(f)$ of each output, shown in Table 5.2:

Table 5.2 – Support and Dependence List of from outputs shown in Equation (5.3).

Output	Support $S(f)$	Dependence $D(f)$
$o0$	$\{i0, i3\}$	$\varnothing$
$o1$	$\{i0, i1, i2, i3, i4\}$	$\{o0, o2, o3\}$
$o2$	$\{i1, i3, i4\}$	$\{o3\}$
$o3$	$\{i1, i4\}$	$\varnothing$

Algorithm 4 FC-FACTOR-HEUR-MO Algorithm

```
1:  function FACTORIZE_MO (outputs, inputs)
2:      if outputs contains 1 output then return FACTORIZE (outputs, inputs)
3:      solutions ← ∅
4:      i ← 0
5:      dep ← ANALYZE_DEPENDENCIES (outputs)
6:      for each output o ∈ outputs do
7:          init ← CREATE_INITIAL_FUNCTIONS (outputs, dep)
8:          solutions[i] ← FACTORIZE (outputs, init)
9:          i ← i + 1
10:     return solutions
```

Algorithm 4 shows the pseudo code for the FC-FACTOR-HEUR-MO. The algorithm needs the circuits outputs ($outputs$) and primary inputs ($inputs$) as arguments. The first step is to check if the circuit is single-output (line 2). In this case, the solution is the factorization using FC-FACTOR-HEUR algorithm. The ANALYZE_DEPENDENCIES method (line 5) will analyze all outputs and extract all dependencies (as presented in Table 5.2). The for loop (line 6-9) will factorize each output, creating a different initial set for each output (line 7). The solutions will be stored and returned at the end of the algorithm.

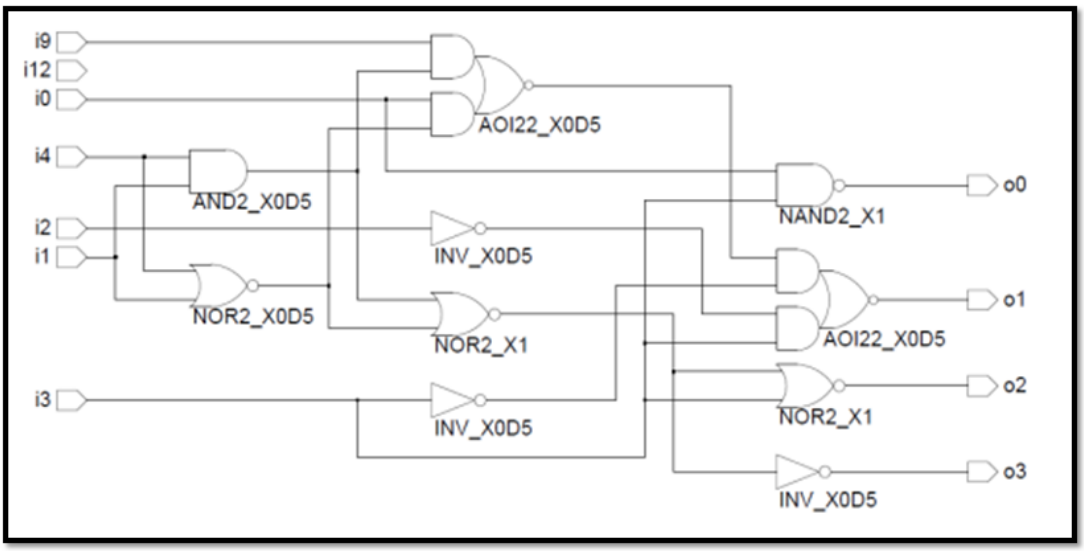
Running the FC-FACTOR-HEUR-MO algorithm in the example and taking advantage of the *"polarity don't care"* inverters present in the example presented in Figure 3.10, we have the following implementations, presented in Equation (5.4). These equations can be

mapped using a DAGON-based technology mapping tool (KEUTZER, 1988) using the double inverter heuristic (DETJENS et al., 1987). The comparison between the original circuit and the resynthesized one is presented in Figure 5.2. The circuit area was reduced in 31% without affecting the timing of the overall circuit. Notice the reduction in the number of the gates (10 to 4), using more complex cells. Also, the 2 inverters ($i9$ and $i12$) were used, reducing the necessary logic in the circuit. One important detail is the cell *SPI7F165*, which implements the logic function $a \cdot b + c \cdot (\mathbf{d} \cdot \mathbf{e} + \mathbf{f} \cdot \mathbf{g})$. This cell implements an unate function. Moreover, it implements a fanout-free or read-once function (i.e. each variable appears only once, which means that there is no logic reconvergence). With the both inverters previously mentioned, The algorithm was able to find an XOR-like function inside a series-parallel function. The bold part is implemented using $i0 \cdot i12 + i1 \cdot i9$, which is logically equivalent to $i0 \cdot \overline{i1} + i1 \cdot \overline{i0}$.

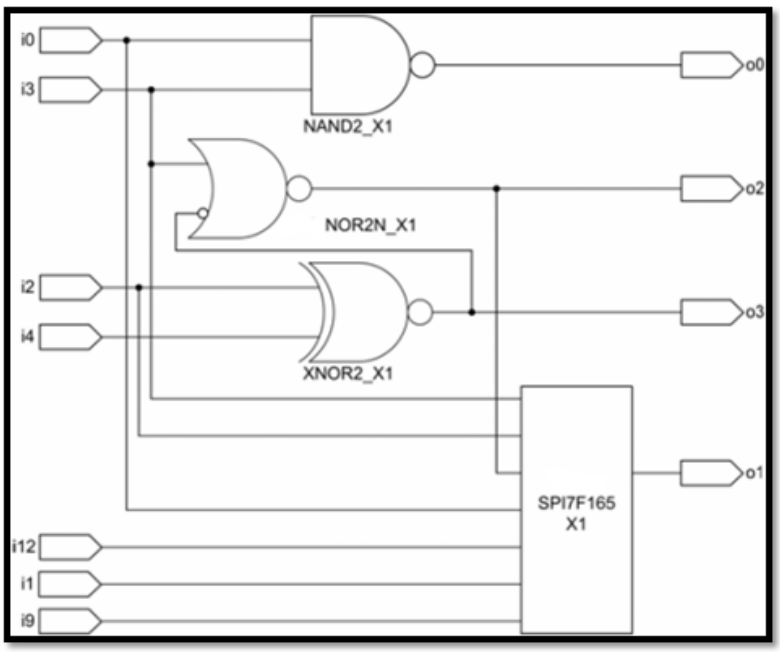
$$
\begin{aligned}
o0 &= \overline{i0} + \overline{i3} \\
o1 &= (i2 \cdot i3) + o2 \cdot (\overline{i0} \cdot i1 + i0 \cdot \overline{i1}) \\
o2 &= o3 \cdot \overline{i3} \\
o3 &= \overline{i1} \cdot \overline{i4} + i1 \cdot i4 \\
i9 &= \overline{i0} \\
i12 &= \overline{i1}
\end{aligned}
\tag{5.4}
$$

5.1.3 Possible Optimizations

The proposed multi-output algorithm requires intensive computation since the circuit can have many cuts and each cut can contain a considerable number of outputs. In this sense, two algorithms specialized in classes of functions are used to speed-up the results. One of them is a synthesis for disjoint-support decomposable (DSD) functions (BERTACCO; DAMIANI, 1997; CALLEGARO et al., 2015; CALLEGARO MAYLER GA MARTINS, 2015). The second one is the synthesis of read-polarity-once (RPO) functions, where each polarity (positive or negative) of a variable appears at most once in the minimum factored expression of an RPO function (ARO et al., 2012; CALLEGARO et al., 2013a; CALLEGARO et al., 2013b; CALLEGARO et al., 2014). These algorithms have two advantages compared to the heuristic factoring: (1) they are faster since they compute only a subset of functions, and (2), they always provide minimal literal count



(a)



(b)

Figure 5.2 – Circuit mapped using a Commercial Tool (a) and synthesized using our resynthesis tool (b).

implementations. In this sense, these algorithms can be used before the heuristic factoring. If they fail to find a solution (i.e. the function is neither RPO or DSD), the heuristic factoring is called upon. Also, the timing can be improved using logic depth or the minimum device chain (MDC) (MARTINS et al., 2011; MARTINS et al., 2011) as second criteria. The MDC of a logic function is related to the maximum number of switches in series in switch networks that implement the given logic function (SCHNEIDER et al., 2005).

5.1.4 Iterative remapping flow

To evaluate the quality of the proposed factorization algorithm, it is necessary an iterative remapping flow. It is important that this flow can be used together with usual commercial flows. After a logic synthesis process using a logic synthesis tool, the result is a gate-level netlist using logic gates of a cell library. This netlist is used as the start point for the proposed iterative remapping approach. On top of this mapped circuit, sub-circuits are found by enumerating KL-cuts (MACHADO et al., 2012). The proposed remapping flow using KL-cuts is shown in Figure 5.3. It is necessary to define the K, i.e. the maximum number of inputs of the sub-circuits derived. The increase of K increases the complexity of the multi-output factorization algorithm, but leads to better results, since more (and larger) sub-circuits are found, and more optimizations can be made. However, the runtime would increase exponentially: more KL-cuts to be remapped and replaced, and more time to factorize the cut is needed.

In this flow, the timing constraints are respected. In this sense, only the KL-cut substitutions that do not impact negatively on the timing constraints are performed. A static timing analysis (STA) tool was implemented to check these timing constraints.

All enumerated sub-circuits are remapped, using as primary cost the number of literals. The factored expressions are applied in a technology mapping algorithm, presented in (CORREIA; REIS, 2004). All remapped sub-circuits are checked if the cost function was improved, and sorted from the largest to the lowest gain. Using a greedy algorithm, the sub-circuits are replaced back in the original circuit in such a way that they do not overlap, and the timing constraints are still respected. This process is repeated while producing gains, or for a limited number of iterations.A greedy selection is applied, presenting good results. Also, by performing the remapping iteratively, there is a significant improvement in the quality of results (MACHADO et al., 2013).

5.1.5 Experimental Results

To validate the iterative flow and the optimizations performed by the proposed multi-output factorization algorithm, a subset of combinational circuits of *IWLS 2005 benchmarks* was mapped with two commercial logic synthesis tools using different libraries (MACHADO et al., 2013). The circuits were mapped with timing constraints, i.e. trying to obtain the best possible area under the timing restrictions were given. The proposed iterative remapping approach was

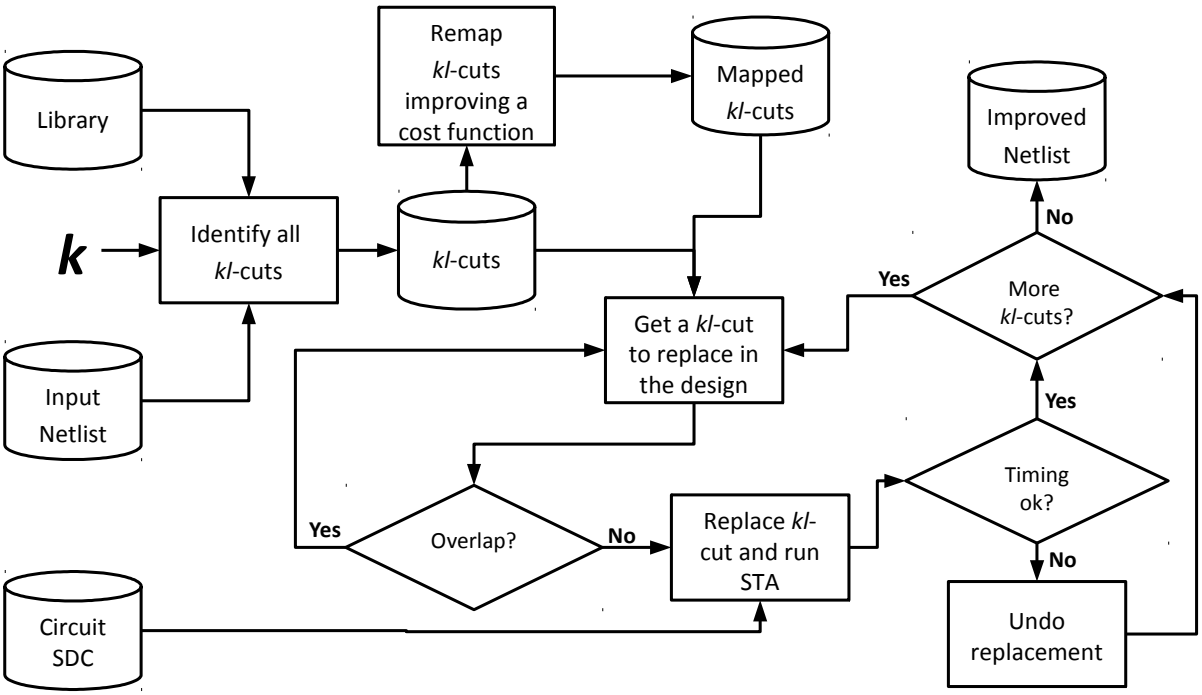


Figure 5.3 – Proposed KL-cut remapping flow (MACHADO et al., 2013).

applied, using a computer with a Core i5 processor and 4GB of RAM. Area is used as the cost function in all the following experiments, with KL-cuts of up to 5 inputs. Two 40nm standard cell libraries were used in the experiments: a base library and an extended library. The base library contains 266 logic gates and a total of 49 different logic functions of combinational cells. The extended library is composed of the base library cells plus all cells with up to three transistors in series and up to three transistors in parallel. In the extended library, there are 132 more logic functions than the base library and a total of 181 different logic functions of combinational cells.

The ITC'99 benchmarks were used in the experiment due to the bigger complexity compared to ISCAS'85 and ISCAS'89 benchmarks. Table 5.3 shows the results obtained by our tool remapping the ITC'99 mapped circuits by commercial tool A. All circuits had area improvement with an average of 17% of combinational area reduced, and the best case of 34% for *b19_1* circuit. Another important data is that the commercial tool A did not deliver the mapped circuit with a timing clean, i.e. the delay was some picoseconds above the delay constraint. Our tool was able to clean timing and improve area using KL-cut remapping. This was a positive collateral effect of the local logic minimization, improving the area and sometimes also improving delay since the number of logic gate stages was decreased.

Table 5.3 – Results obtained with the remapping of commercial logic synthesis tool A mapped circuits, for ITC'99 sequential benchmarks (MACHADO et al., 2013).

Benchmark	Delay constraint (ns)	Remapping of commercial tool A mapped circuits					
		Base Library			Extended Library		
		Comb. Area Diff	Area Diff	Delay(ns)	Comb. Area Diff	Area Diff	Delay(ns)
b01	0.4	-4.55%	-2.02%	0.398	-4.55%	-2.02%	0.398
b02	0.4	-10.87%	-3.25%	0.388	-13.04%	-3.90%	0.357
b03	0.8	-0.40%	-0.09%	0.797	-0.40%	-0.09%	0.797
b04	1.5	-29.15%	-13.50%	1.461	-27.46%	-12.71%	1.393
b05	1.5	-24.04%	-15.15%	1.498	-25.90%	-16.32%	1.487
b06	0.5	-21.30%	-6.55%	0.46	-26.85%	-8.26%	0.432
b07	1	-19.35%	-8.31%	0.928	-21.25%	-9.13%	0.925
b08	0.6	-2.31%	-0.93%	0.598	-4.36%	-1.77%	0.599
b09	1	-10.26%	-2.72%	0.826	-10.26%	-2.72%	0.826
b10	1	-2.56%	-1.18%	0.899	-3.58%	-1.65%	0.9
b11	0.6	-14.71%	-9.99%	0.6	-12.26%	-8.33%	0.599
b12	0.6	-10.79%	-4.89%	0.6	-12.84%	-5.82%	0.599
b13	1	-9.35%	-2.72%	0.704	-9.52%	-2.77%	0.806
b14	2	-21.09%	-17.18%	1.999	-20.52%	-16.71%	2
b14_1	2	-27.93%	-21.06%	1.999	-29.04%	-21.89%	1.999
b15	2	-18.37%	-11.45%	1.998	-18.70%	-11.66%	1.999
b15_1	2	-18.78%	-11.70%	1.997	-19.21%	-11.97%	1.999
b17	2	-18.02%	-11.12%	2	-18.38%	-11.34%	1.999
b17_1	2	-15.90%	-9.77%	2	-17.95%	-11.03%	2
b18	4	-27.53%	-19.86%	4	-27.83%	-20.07%	3.999
b18_1	4	-20.64%	-15.04%	3.999	-31.15%	-22.46%	4
b19	4	-32.77%	-23.69%	4	-32.77%	-23.69%	4
b19_1	4	-33.73%	-24.41%	4	-34.87%	-25.22%	4
b20	2	-11.13%	-9.71%	2	-12.18%	-10.63%	2
b20_1	2	-10.91%	-9.44%	2	-11.52%	-9.97%	2
b21	2	-8.98%	-7.78%	2	-10.51%	-9.11%	2
b21_1	2	-9.60%	-8.29%	2.001	-10.27%	-8.87%	2.002
b22	2	-11.44%	-10.62%	2	-12.37%	-11.43%	2
b22_1	2	-21.07%	-18.26%	2	-22.08%	-19.09%	2
Average	-	-16.12%	-10.37%	-	-17.30%	-11.06%	-
Worst	-	-0.40%	-0.09%	-	-0.40%	-0.09%	-
Best	-	-33.73%	-24.41%	-	-34.87%	-25.22%	-

5.2 Synthesis of Approximate Functions to Mask Transient Faults

Transient faults, such as Single Event Transients (SET), have become a major concern for integrated circuits operating in high-reliability applications. Studies indicate that integrated circuits will be increasingly susceptible to single-event effects (SEE) caused by energetic particles. All these factors have increased the susceptibility of integrated circuits to soft errors (BAUMANN, 2005).

The TMR can mitigate SETs, but they impose a 200% area overhead cost. One strategy to reduce the area overhead is using approximate circuits (SIERAWSKI; BHUVA; MASSENGILL, 2006; ENTRENA et al., 2012; VENKATARAMANI et al., 2012; GOMES et al., 2014). It is also possible to replace three modules of the traditional TMR with approximate circuits, creating a full-ATMR (FATMR) (GOMES et al., 2015a). Using ATMRs allows improvements over TMRs (e.g. performance, power consumption, and area) at the expense of a reduced fault-masking coverage in the presence of upsets.

This section presents an algorithm using FC to compute approximate functions (0-approximate and 1-approximate) efficiently. Previous methods provide one pair of approximate functions or a limited set. By applying functional composition, our approach provides a rich set of approximate functions, which allows a better balance between area and fault coverage. Also, it provides FATMR schemes to reduce the area furthermore, sacrificing a bit more of the protection, whereas previous methods are not able.

The following two subsections present an overview of approximate function, SEE and TMR. Next, we present the proposed FC-based algorithm to synthesize approximate functions. In the sequence, we present the adopted methodology to compose either an ATMR or an FATMR by selecting the most suitable set of functions among those generated by the algorithm in this section. Finally, we present two case studies to certify the applicability of the proposed algorithm together with the adopted methodology.

5.2.1 Approximate Functions

The approximate functions (can also be considered approximate circuits) refers to a class of functions that relax the requirement of exact equivalence between two functions. The concept of approximate functions is related to the order of two functions. Approximate logic functions are defined as functions that may differ from each other in the number of minterms in a Hamming code distance, and it is closely related to the concept of the order. There are two possible types

of approximate functions. Considering G as the original function, one possible approximation is using a smaller function F. The other approximation is considering H as a larger function, compared to G. Figure 5.4 elucidates the $F \subseteq G \subseteq H$ relationship.

Approximate circuits can be used in a TMR scheme, creating an approximate-TMR (ATMR) scheme. However, this imposes a condition on the approximate circuits: only one of the modules can differ from the original circuit at each input vector scenario, allowing the majority voter to select two match outputs out of three for any input vectors. One way to use approximate functions is using a F, G and H as inputs of a majority voter. Any minterm of G must be a minterm of H, and any minterm of F must also be a minterm of G. This relationship ensures that H only evaluates to 0 when G evaluates to 0 and makes F evaluates to 1 when G evaluates to 1. Function H is said to be over-approximated or 0-approximation function. Function F is said to be under approximate or 1-approximation function (SIERAWSKI; BHUVA; MASSENGILL, 2006; ENTRENA et al., 2012).

In Table 5.4 the vector 101 causes $F \neq H$, and G evaluates to 1, if an error occurs in F, causing it to change from 0 to 1, then $F = G = H$. The same idea works for function H, when G evaluates to 0, any error in H will be masked.

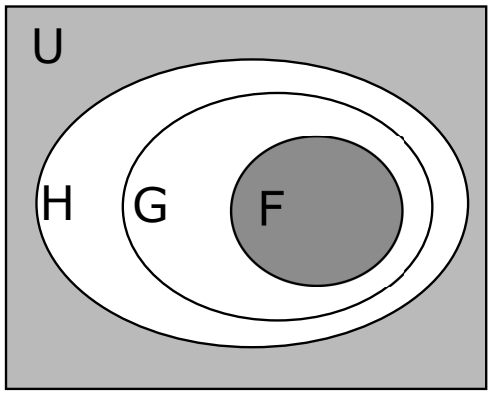


Figure 5.4 – Graphical representation of the relationship between function G, the original function, H, an over approximated (larger) function and F, an under approximated (smaller) function.

The advantage of using approximate circuit is the reduction of the size of the circuit. Therefore, this decreases the overhead of the traditional TMR scheme. However, it may also reduce the fault coverage because there are few of unprotected input vectors.

Also, in some cases, the G function can also be replaced by another F and H, creating a Full ATMR (FATMR) scheme (GOMES et al., 2015a).The most important rule of the FATMR scheme is that only one module may differ from the other two modules to be able to compute the correct value through the majority voter when using spatial redundancy. Therefore, the voter

Table 5.4 – TMR example: Truth table of the approximate circuits for the original function G, where F_1 and H are approximate functions.

Input vectors	$G = A * (B + C)$	$F_1 = A * B$	$H = A$	Voter Output
000	0	0	0	0
001	0	0	0	0
010	0	0	0	0
011	0	0	0	0
100	0	0	**1**	0
101	1	**0**	1	1
110	1	1	1	1
111	1	1	1	1

will have two correct outputs against the one divergent output. For example, a FATMR can be formed of F_1, F_2, and H. For instance, F_1 and F_2 are both under-approximate, but they diverge from G in a different set of minterms. An FATMR cannot be composed of two functions of the same type that differ from function G by the same set of input vectors. This would result in an incorrect value of the majority voter output in the absence of faults. This further reduces the area but sacrifices more the protection. An example of FATMR is shown in Table 5.5.

Table 5.5 – Truth table for a FATMR composed of F_1, F_2 and H_1.

Input vectors	G	F_1	F_2	H_1	Voter output
000	$\underline{0}$	0	0	0	$\underline{0}$
001	$\underline{0}$	0	0	0	$\underline{0}$
010	$\underline{0}$	0	0	0	$\underline{0}$
011	$\underline{0}$	0	0	0	$\underline{0}$
100	$\underline{0}$	0	0	**1**	$\underline{0}$
101	$\underline{1}$	**0**	1	1	$\underline{1}$
110	$\underline{1}$	1	**0**	1	$\underline{1}$
111	$\underline{1}$	1	1	1	$\underline{1}$

5.2.2 Single Event Effects Overview

Malfunctions can occur in integrated circuits due to radiation effects from high energy as neutrons or alpha particles. The malfunction derived from radiation affect mainly aerospace applications as satellites and probes, as well as airplane control and communication modules. As the transistor continue to shrink its dimensions, the supply voltage also reduces, increasing the possibility of the circuit being affected by a particle strike. If a collision of an energetic particle produces a transient channel between the junction and the substrate, a current pulse is

generated. Figure 5.5 illustrates this collision. A single event effect (SEE) is when a particle collision causes a momentary state change of a circuit node, first postulated by Wallmark and Marcus (WALLMARK; MARCUS, 1962). If this event occurs in a memory cell, as a flip-flop or a register, it can modify the original value. This change is usually called bit-flip and is considered a single event upset (SEU). However, if this event occurs in a combinational cell, the effect is called single event transient (SET) since a transient pulse can be generated and propagated through the combinational logic, reaching into a sequential element (BAUMANN, 2005). Not all SETs are fully propagated since the SET can be logically masked, electrically masked and masked by the clock window. One of the ways to protect the combinational logic from SETs is using the concept of the triple modular redundancy (TMR).

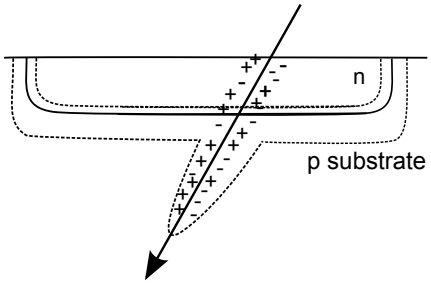


Figure 5.5 – Ionization track caused by a particle strike.

5.2.2.1 Triple Modular Redundancy

The triple modular redundancy was a concept to increase reliability proposed by von Neumann in 1956 (NEUMANN, 1956). The idea is triple the circuits and uses a majority-voting system to decide the correct result. A TMR example is shown in Figure 5.6. This redundancy allows one circuit to fail temporarily or permanently. A source of a temporary fail is a SET, which will be fully masked by the majority voter. Two drawbacks for this scheme are the area and power overhead of more than 200%.

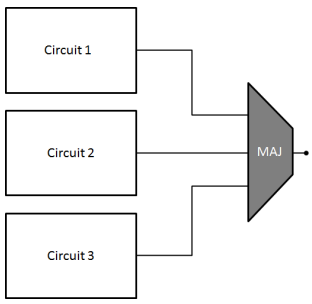


Figure 5.6 – Example of a TMR system.

5.2.3 Synthesis of Approximate Functions

The algorithm FC-FATMR described in Algorithm 5 computes the approximate functions using functional composition, using a slightly modified version of FC-FACTOR-HEUR, already presented in Algorithm 3. The main difference is that the after the function is factored ($Gexp$), the solution also includes the list of factored smaller ($Flist$) and larger ($Hlist$) functions. Also, it computes all possible FATMR combinations from the approximate functions, through the COMPOSE method, which implements a majority function functionally. All 4 possible FATMR schemes are covered: 3 F functions, 3 H functions, 2 F functions and 1 H function, and 2 H functions and 1 F function. Another difference is the inclusion of the constant *zero* as a allowed function (smaller) and the constant *one* (larger). These functions can be useful as candidates in functions that have a small (large) number of minterms.

Algorithm 5 FC-FATMR Algorithm

```
1:  function CREATE_APPROXIMATE_FUNCTIONS (G)
2:      {Gexp, Flist, Hlist} ← FACTORIZE_TMR (G)
3:      FATMRlist ← ∅
4:
5:      for each function f1, f2, f3 ∈ Flist do
6:          fatmr ← COMPOSE (f1, f2, f3)
7:          if fatmr = G then FATMRlist ← FATMRlist ∪ fatmr
8:
9:      for each function h1, h2, h3 ∈ Hlist do
10:         fatmr ← COMPOSE (h1, h2, h3)
11:         if fatmr = G then FATMRlist ← FATMRlist ∪ fatmr
12:
13:     for each function f1, f2 ∈ Flist do
14:         for each function h1 ∈ Hlist do
15:             fatmr ← COMPOSE (f1, f2, h1)
16:             if fatmr = G then FATMRlist ← FATMRlist ∪ fatmr
17:
18:     for each function h1, h2 ∈ Hlist do
19:         for each function f1 ∈ Flist do
20:             fatmr ← COMPOSE (h1, h2, f1)
21:             if fatmr = G then FATMRlist ← FATMRlist ∪ fatmr
22:
23:     return {Gexp, Flist, Hlist, FATMRlist}
```

5.2.4 Approximate Circuits Methodology

The methodology used (GOMES et al., 2015a) selects the most suitable set of functions F and H able to compose an ATMR or FATMR, generated by the FC-FATMR method. The functions are mapped through an academic state-of-the-art logic synthesis tool (BRAYTON; MISHCHENKO, 2010), in conjunction with a library, containing the following cells: INV, NAND2-3-4, NOR2-3-4, OAI21, OAI22, AOI21 and AOI22, a subset from (ALBRECHT, 2005). Faults are injected in the circuits using a tool developed to generate a transistor level description of the circuits (including the protection using ATMR or FATMR) and test the circuits susceptibility to single event transients (SET) (GOMES et al., 2015a). A logic XOR between G and the candidate function is performed to compute the Hamming distance between approximate functions. In other words, the first criteria are the number of unprotected vectors that can be used to estimate early the fault masking coverage and the number of literals that can be used to estimate the area early.

An ATMR design tool is used to generate the transistor level circuit of each ATMR and FATMR scheme composed(GOMES et al., 2015a). The tool receives a set of ATMR/FATMR compositions, automatically map those schemes to a gate netlist and later generate a transistor level description in Verilog, for each composition.

The evaluation of the fault masking capability of each ATMR and FATMR scheme is performed through a fault injection tool (GOMES et al., 2015a). The sizing of the transistor, electrical masking (signal is attenuated or eliminated by the electrical properties of gates), and temporal masking (the erroneous pulse reaches a flip-flop, but it is not captured) are not evaluated in this type of fault model. In this sense, each redundant module is simulated in SPICE to evaluate how transistor sizing, electrical and temporal masking would affect the fault masking coverage of the ATMR/FATMR scheme (GOMES; KASTENSMIDT et al., 2013).

The approximate synthesis method was applied in the methodology above presented, and the quality of approximate functions was tested in two case studies, a simple 5 input function with 10 literals and a 4-bit ripple carry adder. Each of the main functions was used to create several different ATMR schemes. The approximate functions were synthesized using FC-FATMR algorithm. The factored form was implemented in a Verilog file and mapped using the ABC tool (Berkeley Logic Synthesis and Verification Group, 2013). The mapped netlists were tested for SET using the ModelSim.

5.2.5 Case-study Circuit 1: 5-input Boolean Function

The first case of study was created based on the 5 input function represented by Equation (5.5):

$$G = (\overline{a} + \overline{b}) \cdot ((\overline{c} \cdot b) + ((a + e) \cdot ((a \cdot (c + e)) + d))) \tag{5.5}$$

Table 5.6 shows a list of selected approximate functions F and H for the case-study circuit 1. Table 5.7 shows the characteristics of each ATMR and FATMR schemes composed of different selected approximate functions regarding unprotected vectors, fault masking coverage, and area overhead. The schemes in the table are ordered increasingly using the area overhead as order criteria. Notice that each scheme has a different number of unprotected vectors and unprotected junctions. The highest fault masking coverage is given by the schemes that present the lowest number of unprotected vectors and junctions. The area is measured by the number of transistors.

Table 5.6 – Functions F and H for the case-study circuit 1 (GOMES et al., 2015a).

	Approximate functions
F1	$\overline{a} \cdot b \cdot \overline{c}$
F2	$a \cdot \overline{b} \cdot (d + e)$
F3	$a \cdot \overline{b} \cdot d + \overline{a} \cdot e$
F4	$(\overline{a} + \overline{b}) \cdot (\overline{c} \cdot b + d \cdot e)$
F5	$(\overline{a} + \overline{b}) \cdot ((a + e) \cdot (a \cdot (c + e)) + d)$
H1	$\overline{a} + \overline{b}$
H2	$\overline{b} + (\overline{a} \cdot (\overline{c} + e))$
H3	$a \cdot \overline{b} + (\overline{a} \cdot (\overline{c} + e))$
H4	$a \cdot \overline{b} + (\overline{a} \cdot (\overline{c} + e \cdot d))$
H5	$d \cdot e + (a \cdot \overline{b} + (\overline{c} \cdot \overline{a} \cdot b))$
H6	$d \cdot \overline{a} \cdot e + a \cdot \overline{b} + \overline{c} \cdot \overline{a} \cdot b$

Table 5.7 – Functions G, F and H for the full-adder and half adder (GOMES et al., 2015a).

Functions	Full-adder	Half-adder
$Gsum$	$a \oplus b \oplus Cin$	$a \oplus b$
$Gcout$	$(a \cdot b) + (Cin \cdot (a \ominus b))$	$a \cdot b$
$Fsum$	$a \cdot b \cdot Cin$	$\overline{\overline{a} + b}$
$Fcout$	$a \cdot b$	0
$Hsum$	$a + b + Cin$	$\overline{a \cdot b}$
$Hcout$	$a + b$	a

Table 5.8 – Characteristics of the ATMR and FATMR schemes for the Case-study 1 (5 input, 10 literal's function) (GOMES et al., 2015a).

Different Scheme Implementations of Circuit G	Compositions	# Unprotected vectors / Total vectors	% of Protected input vectors	# Unprotected p-n Junctions / # Total p-n junctions	% of Protected Junctions (Masking Coverage)	# transistors (Estimated area)	Area Overhead
Single-module G	G/ - / -	32/32	0%	**228/2048**	88.86%	32	-
FATMR 1	F1/F5/H1	23/32	28.12%	209/2304	90.93%	36	13%
FATMR 2	F1/F5/H2	21/32	37.50%	199/2816	92.93%	44	38%
ATMR 1	G/F1/H1	20/32	34.37%	237/2816	91.58%	44	38%
FATMR 3	F1/F5/H3	19/32	40.62%	212/3072	93.10%	48	50%
ATMR 2	G/F2/H1	18/32	46.87%	171/3072	94.43%	48	50%
FATMR 4	F1/F5/H4	17/32	53.12%	201/3328	93.96%	52	63%
ATMR 3	G/F3/H1	16/32	43.75%	184/3584	94.87%	56	75%
FATMR 5	F1/F5/H5	16/32	56.25%	195/3712	94.75%	58	81%
FATMR 6	F1/F5/H6	14/32	50.00%	205/3840	94.66%	60	88%
ATMR 4	G/F4/H1	15/32	50.00%	223/3840	94.19%	60	88%
ATMR 5	G/F5/H1	13/32	59.37%	147/3840	96.17%	60	88%
ATMR 6	G/F5/H2	11/32	65.62%	139/4352	96.81%	68	113%
ATMR 7	G/F5/H3	9/32	71.87%	120/4608	97.40%	72	125%
ATMR 8	G/F5/H4	7/32	78.12%	91/4864	98.13%	76	138%
ATMR 9	G/F5/H5	6/32	81.25%	76/5248	98.55%	82	156%
ATMR 10	G/F5/H6	4/32	87.50%	60/5248	98.88%	84	163%
TMR	G/G/G	0/32	100%	0/6144	100%	96	200%

One can observe that it is possible to maintain the maximum protected p-n junction ratio of 98.88% (60 p-n junctions of 5248) with only 165% area overhead when using ATMR; and a maximum of 94.66% protected p-n junction ratio (205 p-n junctions of 3840) with only an 88% area when using FATMR.

5.2.6 Case-study Circuit 2: 4-bit Ripple Carry Adder

The second case of study is an ATMR design of a 4-bit adder. The 4-bit adder is composed by one half-adder at bit 0 and other 3 full-adders. Table 5.7 shows the approximate logic functions used for the Sum and Cout outputs for both cases.

The original full-adder (G) used has a size of 28 transistors (complex gate). The under-approximate full-adder (F) has a size of 12 transistors (2 NAND2 gates and 2 inverters). The over-approximate full-adder (H) has a size of 12 transistors (2 NOR2 gates and 2 inverters).

The original half-adder (G) has a size of 16 transistors (1 XOR2, 1 AND2, and 1 inverter). The under-approximate full-adder (F) has a size of 6 transistors (1 NOR2 gate and 1 inverter). The over-approximate full-adder (H) has a size of 4 transistors (1 NAND2 gate).

Each ATMR scheme is composed of 12 different modules separated into 4 levels (bit 0 corresponds to level 1 and bit 3 corresponds to level 4). Each level has 3 modules (G, F, and H). The sum bit is evaluated by voting $Gsum$, $Fsum$ and $Hsum$ of each level, as seen in Figure 5.7. Moreover, the carry out is evaluated by voting $Gcout$, $Fcout$ and $Hcout$ of level 4.

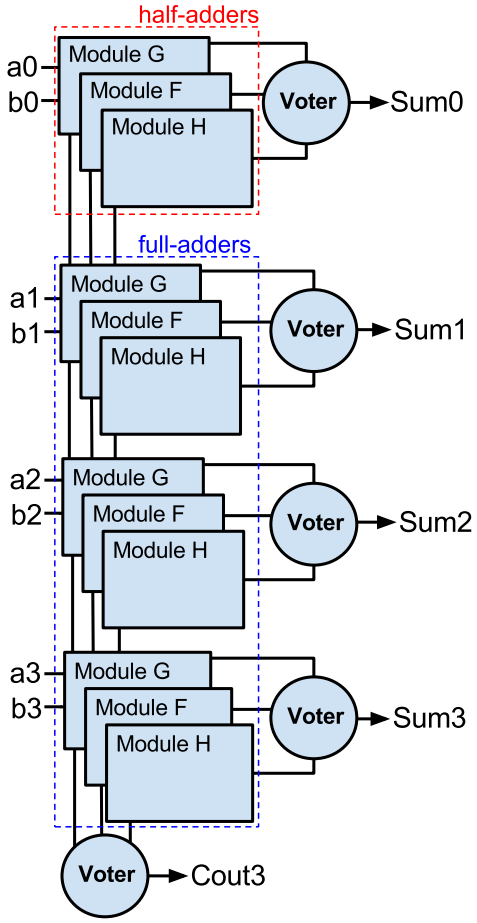


Figure 5.7 – ATMR scheme for a 4-bit adder (GOMES et al., 2015a).

Table 5.9 shows how each of the tested ATMR schemes is composed regarding the modules in each level of the design for sum and carry outputs The schemes in the table are ordered increasingly by the number of transistors. One can observe that some of the ATMR present a higher number of unprotected p-n junction compared to the single module circuit (the adder with no TMR).

The trade-off between masking and area overhead is only attractive from designs ATMR5 to ATMR11, as they present a lower number of unprotected p-n junction compared to the single module circuit (the adder with no TMR).

Also it is important to note that the best way to achieve a good trade-off between area overhead and masking coverage is to use approximate modules by level, for example, both

ATMR6 and ATMR7 have 152% of area overhead, and both uses 3 approximate modules, but ATMR6 is better than ATMR7, the difference between them is how the approximate modules are spread. ATMR6 uses two approximate module in level 4 and one in level 3 of the circuit, ATMR7 allocates one module for each level, starting from level 2 to level 4. Something similar can be seen when we compare ATMR6 to ATMR5, were ATMR5 has 4 approximate module, two for level 3 and two for level 4, showing a trade-off when compared to ATMR6.

Table 5.9 – Characteristics of the ATMR schemes for the Case-study 2 (4-bit ripple-carry adder) (GOMES et al., 2015a).

Different Schemes of the 4-bit adder	Compositions				# Unprotected vectors / Total vectors	# Unprotected p-n Junctions	% of Protected Junctions (Masking Coverage)	# transistors (Estimated area)	Area Overhead
	$lvl.1$	$lvl.2$	$lvl.3$	$lvl.4$					
Single G	$G/-/-$	$G/-/-$	$G/-/-$	$G/-/-$	256/256	**9920**	80.63%	100	0%
ATMR1	$G/F/H$	$G/F/H$	$G/F/H$	$G/F/H$	128/256	11424	87.74%	184	82%
ATMR2	$G/G/G$	$G/F/H$	$G/F/H$	$G/F/H$	126/256	10320	90.12%	204	104%
ATMR3	$G/G/H$	$G/G/H$	$G/F/H$	$G/F/H$	128/256	10664	89.99%	208	108%
ATMR4	$G/G/G$	$G/F/G$	$G/F/H$	$G/F/H$	125/256	9976	91.14%	220	120%
ATMR5	$G/G/G$	$G/G/G$	$G/F/H$	$G/F/H$	120/256	7912	93.45%	236	136%
ATMR6	$G/G/G$	$G/G/G$	$G/F/G$	$G/F/H$	116/256	7476	94.21%	252	152%
ATMR7	$G/G/G$	$G/F/G$	$G/F/G$	$G/F/G$	123/256	9344	92.76%	252	152%
ATMR8	$G/G/G$	$G/G/G$	$G/F/G$	$G/F/G$	112/256	7136	94.80%	268	168%
ATMR9	$G/G/G$	$G/G/G$	$G/G/G$	$G/F/H$	96/256	5108	96.28%	268	168%
ATMR10	$G/G/G$	$G/G/G$	$G/G/G$	$G/F/G$	80/256	4488	96.91%	284	184%
ATMR11	$G/G/G$	$G/G/G$	$G/G/G$	$G/G/H$	80/256	4616	96.83%	284	184%
TMR	$G/G/G$	$G/G/G$	$G/G/G$	$G/G/G$	0/256	0	100%	300	200%

6 EMERGING TECH. APPLICATIONS USING FUNCTIONAL COMPOSITION

In this chapter, we explore the synthesis of emerging technologies using functional composition. The first application presented in this chapter explores the synthesis of threshold logic, which can be applied both for asynchronous approaches (MOREIRA et al., 2014), as also for RTD and STT-MTJ devices (NEUTZLING et al., 2013b; NEUTZLING et al., 2013; NEUTZLING et al., 2014; NEUTZLING et al., 2014; NEUTZLING et al., submitted). The second application does a specialized threshold logic synthesis, focused on majority gates for QCA, SET, TPL and other majority-based devices (MARTINS et al., 2014; MARTINS et al., 2014). The third application proposes an automated flow for spin-diode circuits (MARTINS et al., 2013; MARTINS et al., 2015a; MARTINS et al., 2015b) and the last application discusses synthesis of factored forms for memristor sequential logic (MARRANGHELLO et al., 2014b; MARRANGHELLO et al., 2014a; MARRANGHELLO et al., 2015b; MARRANGHELLO et al., 2015a).

6.1 Synthesis of Threshold Logic for Emerging Technologies

To exploit the advantages of threshold logic in new technologies, EDA tools that automate the design of integrated circuits directly on this logical style are required. There is an extra interest in the study of threshold circuits because networks constructed with threshold gates are almost equivalent to standard feed forward neural networks models, as discussed in Section 2.2.1. In this sense, most of the properties and characteristics of the circuits can be extended and applied to neural networks (BEIU; QUINTANA; AVEDILLO, 2003; SUBIRATS; JEREZ; FRANCO, 2008).

In this section, FC will be used to synthesize threshold logic efficiently. Two algorithms are proposed: One synthesize minimal threshold networks for functions up to 4 inputs, and the other algorithm uses heuristics to synthesize functions up to 6 inputs, reducing the number of threshold logic gates in a circuit furthermore.

6.1.1 Synthesis of Threshold Networks

To synthesize efficiently threshold networks, an effective AND/OR association of threshold networks is necessary. In (NEUTZLING et al., 2014), it is presented an effective way to associate two threshold networks, which is demonstrated in Figure 6.1.

To illustrate the association, let $f_1(x_1, x_2, x_3) = x_1 \cdot x_2 + x_1 \cdot x_3$ that is a TLF represented by $[2, 1, 1; 3]$. Then, $h_1(x_1, x_2, x_3, x_4) = (x_1 \cdot x_2 + x_1 \cdot x_3) \wedge x_4$ is also TLF represented by $[2, 1, 1, 2; 5]$, since $T_h = 1 + \sum(w_1, w_2, w_3) = 5$ and $w_4 = T_h - T_f = 2$.

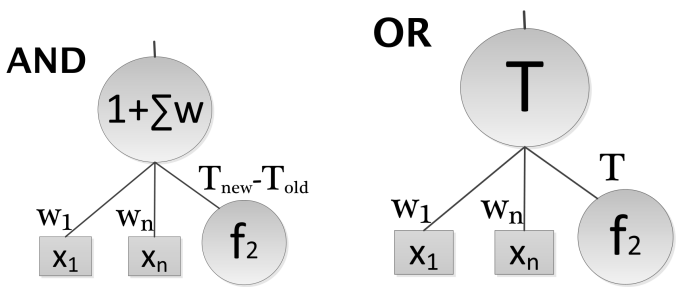


Figure 6.1 – An efficient way to associate threshold networks (NEUTZLING et al., 2014).

6.1.1.1 Optimal 4-input threshold network generation

Optimal TLG implementations containing all functions up to 4 inputs can be easily generated with a straightforward procedure, using FC. The algorithm is presented in Algorithm 6. This approach is attractive to a mapping point-of-view, since it is necessary only one execution to generate a full library, and the results are stored for later reuse, avoiding the matching task. The algorithm needs the set of all unate functions synthesized in 1 TLG ($init$) and the number of variables (n). The algorithm combines the TLGs until all functions up to n inputs are implemented.

The direct and negated variables are stored in bucket 0 (line 3), since they do not have gate implementation costs. The next task is getting all functions that can be implemented as a single TLG. All unate functions up to n variables ($n \geq 4$) can be provided by the identification algorithm to determine which function is TLF (NEUTZLING et al., 2013b; NEUTZLING et al., 2013; NEUTZLING et al., 2013a) and inserted in Bucket 1 (line 4). The ASSOCIATE method (line 10-16) performs all combinations, considering TLG count as the primary cost.

Algorithm 6 FC-TLG-EXACT Algorithm

```
1:  function CREATE_ALL_FUNCTIONS_TLG (n, init)
2:      i ← 1
3:      B[0] ← CREATE_INITIAL_FUNCTIONS ()
4:      B[1] ← init
5:      while any function is not synthesized do
6:          B[i] ← ASSOCIATE (B, i)
7:          i ← i + 1
8:      return B
9:
10: function ASSOCIATE (B, i)
11:     S ← ∅
12:     for k ← 0, (i/2) do
13:         l ← i − k
14:         S ← S ∪ COMBINE (B[k], B[l], AND)
15:         S ← S ∪ COMBINE (B[k], B[l], OR)
16:     return S
```

6.1.1.2 Threshold network synthesis up to 6 inputs

Unfortunately, the universe of 4-input functions is very limited in comparison to one of the 6-input functions. Therefore, it is important to be available an algorithm that synthesizes, even heuristically, functions with 5 and 6 inputs. In this sense, a heuristic threshold factoring algorithm using FC is proposed. The algorithm FC-TLG-HEUR is presented in Algorithm 7. The heuristic applied is based on the FC-FACTOR-HEUR, presented in Section 5.1.1.2.

Since a combination of two functions can originate a TLF, this will jeopardize the partial order. In this sense, a modification in the method COMBINE_COFACTORS is necessary. Also, if the function presents 4 inputs or less, the information from the $Catalog$ comprising optimal 4-input threshold network can be accessed.

The first step for the synthesis of threshold network up to 6 inputs is to check if the target function is TLF (line 2). If this condition is attained, the algorithm returns a TLG provided by the identification algorithm (NEUTZLING et al., 2013b; NEUTZLING et al., 2013; NEUTZLING et al., 2013a). The second step is checking if the support of the function is less than 5 inputs (line 3). If the condition is satisfied, the algorithm uses the catalog and return the minimal implementation. Another difference is the COMBINE_COFACTORS_TLG method (line 7), which uses the catalog to check the result of each combination. If the support of the combined function is in the catalog, the implementation is retrieved and inserted in the right Bucket. If no implementation is found (i.e. the cofactor has support greater than 4), the identification algorithm is executed. These modifications guarantee all minimum cost functions were identified before the combination loop (line 9-11), avoiding a violation of the partial order. The AND and OR associations are the same from Figure 6.1. Moreover, multiple cost functions using the threshold parameters (gate count, logic depth, and number of interconnections) can be used to select the best implementation.

6.1.2 Experimental Results

The experimental results evaluate the efficiency of the both methods for threshold network synthesis. Figure 6.2 illustrates the synthesis flow for circuits. The platform used in the experiments was an Intel Core i5 processor with 2 GB main memory. FC-TLG-EXACT and FC-TLG-HEUR have the following cost function, in this order of priority: (1) threshold gate count, (2) logic depth and (3) number of interconnections.

Algorithm 7 FC-TLG-HEUR Algorithm

```
1:  function SYNTHESIZE_TLG (f, Catalog)
2:      if IDENTIFY (f) then return solution
3:      if SUPPORT (f) ≤ 4 then return Catalog (f)
4:      B ← ∅
5:      B[1] ← CREATE_INITIAL_FUNCTIONS (f)
6:      cofactors ← COMPUTE_COFACTORS (f)
7:      allowed_functions ← COMBINE_COFACTORS_TLG (cofactors, Catalog, B)
8:      i ← 2
9:      while f is not found do
10:         B[i] ← COMBINE_BUCKETS(B, i)
11:         i ← i + 1
12:     return solution
```

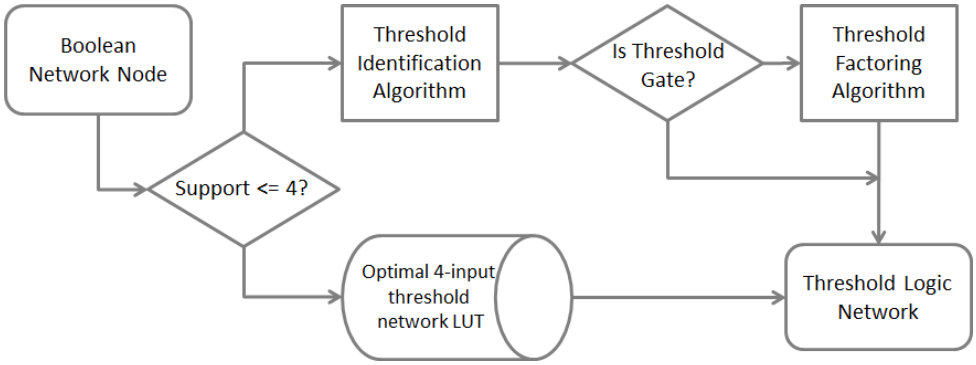


Figure 6.2 – Function synthesis flow (NEUTZLING et al., 2014).

MCNC benchmark circuits (YANG, 1991) were decomposed and mapped to compare our results to the ones presented in (ZHANG et al., 2005) and in (GOWDA et al., 2011). Figure 6.2 shows the design flow applied to synthesize the circuits. To a fair comparison, SIS tool (SENTOVICH et al., 1992) was used to decompose the circuits. Since Zhang et al., in (ZHANG et al., 2005), and Gowda et al., in (GOWDA et al., 2011), do not mention which SIS scripts they applied to decompose the circuits, the scripts from (SUBIRATS; JEREZ; FRANCO, 2008) were chosen and adapted to generate networks with up to 6 inputs. We synthesized all circuits listed in (ZHANG et al., 2005), and we were able to reduce the threshold gate count in all MCNC benchmarks evaluated. However, for the sake of simplicity, we present only the 20 more relevant circuits, which were implemented using more than 70 threshold gates. The decomposition results obtained using ABC tool (Berkeley Logic Synthesis and Verification Group, 2013) were also omitted, although gains have been verified. Circuits synthesized using ABC generated an increasing around 26.4% in the gate count, with a significant reduction in the logic depth, of circa 50%.

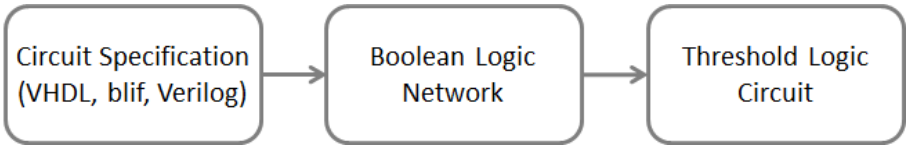


Figure 6.3 – Threshold logic circuit synthesis flow (NEUTZLING et al., 2014).

Table 6.1 and Table 6.2 shows the results of mapping threshold networks in MCNC benchmarks and presents the gate count reduction in the evaluated circuits, demonstrating the efficiency of the proposed method. The average gate count reduction is of circa 32%, reaching up to 54%. Nevertheless, many circuits achieved some improvement in the three characteristics simultaneously. We were also able to reduce or maintain the logic depth in the circuits presented in Table I, except for the *pair* and *des* benchmarks. In average, the logic depth decreased 19.3%. An increasing in the number of interconnection was expected since the method tries to use a maximum fan-in always when it is possible. In general, this cost has also been improved due to the multi-goal synthesis, prioritizing the threshold gate count, logic depth and number of interconnections, in this order.

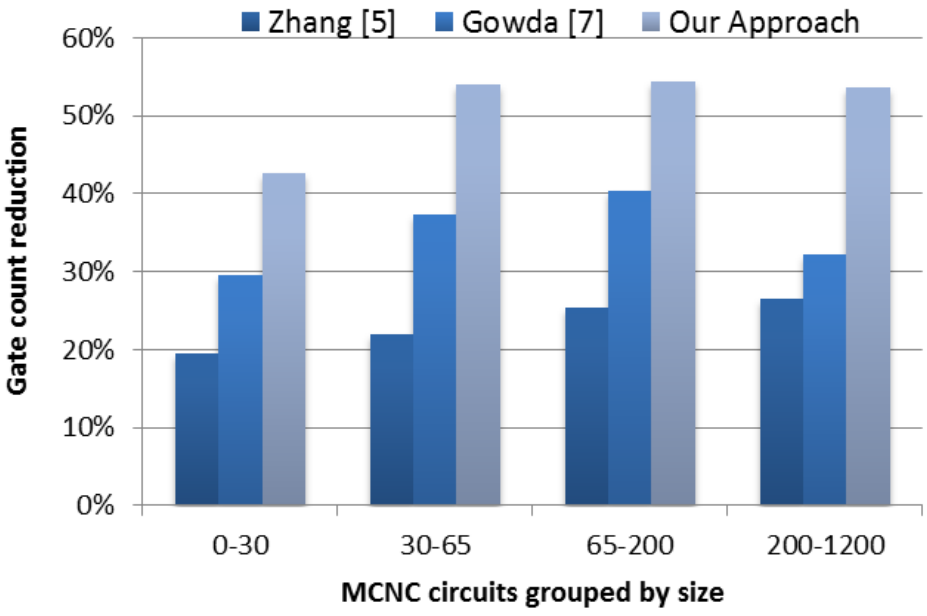


Figure 6.4 – Percentage gate count reduction in each approach, compared to the original netlist (ZHANG et al., 2005).

The results presented by Gowda et al., in (GOWDA et al., 2011), show an improvement in threshold gate count compared to the results provided in (ZHANG et al., 2005). However, in (GOWDA et al., 2011), the authors only compare the gate count and present the results for MCNC circuits grouped by number of inputs.Figure 6.4 shows the gate reduction of each

Table 6.1 – Results presented in (NEUTZLING et al., 2014) using MCNC benchmarks with more than 25 inputs, compared to (ZHANG et al., 2005).

Benchmark	Inputs	Outputs	G	L	I	G%	L%	I%
i10	257	224	840	31	3072	53.77	11.43	47.87
des	256	245	1556	19	5210	18.96	-18.75	-0.58
i2	201	1	62	6	353	68.69	14.29	49.14
i7	199	67	197	3	925	35.20	40.00	-13.78
i4	192	6	70	9	256	5.41	-80.00	23.81
pair	173	137	563	17	2141	37.93	-41.67	27.82
i6	138	67	141	3	656	48.91	40.00	0.30
x3	135	99	280	7	1125	36.51	0.00	25.69
apex6	135	99	279	10	1098	29.55	16.67	6.07
i8	133	81	427	10	1330	25.09	0.00	25.70
i5	133	66	66	5	324	0.00	16.67	-24.62
i3	132	6	86	5	212	45.57	16.67	54.31
x4	94	71	152	5	522	19.58	37.50	7.12
i9	88	63	266	8	951	3.27	0.00	-16.40
example2	85	66	151	6	538	17.03	25.00	-10.02
dalu	75	16	371	11	1485	54.20	52.17	42.42
x1	51	35	107	5	427	47.29	28.57	41.59
apex7	49	37	78	7	322	33.90	22.22	11.54
cht	47	36	73	2	237	10.98	60.00	-17.33
unreg	36	16	48	2	176	4.00	60.00	-31.34
count	35	16	55	11	206	30.38	8.33	14.52
term1	34	10	60	7	245	73.45	30.00	64.13
my adder	33	17	71	10	247	26.04	44.44	18.75
comp	32	3	35	8	125	57.83	0.00	59.81
c8	28	18	58	5	196	31.76	28.57	14.04
frg1	28	3	36	8	154	38.98	11.11	33.91
pcler8	27	17	36	4	128	23.40	42.86	10.49
lal	26	19	32	4	142	40.74	42.86	15.48
TOTAL AVERAGE REDUCTION:						**32.80**	**18.18**	**17.16**

*G=gates, L=logic depth, I= #of interconnections, %G, %L, %I = reduction.

approach, compared to the original netlist (ZHANG et al., 2005), which uses only the traditional OR and AND description.

The graphic shown in Figure 6.4 demonstrates that the reduction in gate count is larger than the reduction presented in (ZHANG et al., 2005). However, in (GOWDA et al., 2011), which has been considered in this work as the state-of-art threshold network synthesis approach. The proposed method has provided an average reduction of 51.2% in comparison to the original netlist, against a reduction of 23.3% and 34.8% obtained in (ZHANG et al., 2005). However, in (GOWDA et al., 2011), respectively.

Table 6.2 – Results presented in (NEUTZLING et al., 2014) using MCNC benchmarks with 25 inputs or less, compared to (ZHANG et al., 2005).

Benchmark	Inputs	Outputs	G	L	I	G%	L%	I%
i1	25	16	14	4	49	39.13	20.00	22.22
ttt2	24	21	62	6	256	38.00	0.00	21.71
cordic	23	2	24	6	78	31.02	14.29	49.68
cc	21	20	23	3	71	34.29	50.00	21.98
cm150a	21	1	21	5	72	0.00	-25.00	6.49
pcle	19	9	27	4	104	22.86	33.33	4.59
sct	19	15	25	11	93	34.21	-120.00	19.13
tcon	17	16	16	2	40	50.00	33.33	28.57
parity	16	1	30	8	75	33.33	11.11	16.67
pm1	16	13	16	4	58	34.43	0.00	23.68
cm163a	16	5	15	5	56	40.00	16.67	33.33
cmb	16	4	13	4	62	51.85	33.33	12.68
alu4	14	8	275	22	1112	32.93	4.35	20.97
cu	14	11	17	3	66	29.17	25.00	13.16
cm162a	14	5	15	5	58	42.31	37.50	34.09
cm151a	12	2	11	5	35	8.33	0.00	22.22
cm152a	11	1	10	4	33	9.09	0.00	21.43
cm85a	11	3	8	3	44	42.86	40.00	38.89
alu2	10	6	134	18	307	31.98	28.00	29.09
x2	10	7	13	4	52	13.33	0.00	22.39
9symml	9	1	23	7	111	79.09	22.22	73.06
f51m	8	8	24	6	118	70.73	25.00	55.64
z4ml	7	4	12	4	51	36.84	20.00	20.31
decod	5	16	16	1	80	33.33	66.67	-53.85
cm82a	5	3	8	3	37	33.33	25.00	2.63
majority	5	1	1	1	5	0.00	50.00	0.00
cm42a	4	10	10	1	40	23.08	66.67	-17.65
b1	3	4	5	2	13	37.50	33.33	18.75
TOTAL AVERAGE REDUCTION:						**33.89**	**18.24**	**20.07**

*G=gates, L=logic depth, I= #of interconnections, %G, %L, %I = reduction.

6.2 Synthesis of Majority Logic for Emerging Technologies

The direct transformation of AND and OR operations to a majority expression is not optimal for majority technologies (ZHANG et al., 2004). For this reason, several works have proposed specific methods to obtain better majority-based expressions and, consequently, optimized circuit implementations (ZHANG et al., 2004; MOMENZADEH et al., 2005; KONG; SHANG; LU, 2010; WANG et al., 2013).

In this section, we propose a method using FC able to synthesize majority logic for QCA, SET, and TPL. Furthermore, we use a special AOI gate, which allows a considerable area reduction for the QCA technology.

6.2.1 Related Work

The majority-based logic synthesis can be considered as a sub-area of the threshold logic synthesis, which dates back to 1960's with the pioneering works of Akers (AKERS, 1962), Miller and Winder (MILLER; WINDER, 1962) and Muroga (MUROGA, 1971). However, these works describe procedures that are only suitable to synthesize small circuits by hand.

In (ZHANG et al., 2004), Zhang et al. considered a set of 13 NPN classes for 3-input functions. NPN class consists of functions that are not functional equivalent even doing inputs and output negation and inputs permutation. The authors proposed a majority-based implementation for each one of these classes, showing that the direct transformation does not lead to the optimal solution.

In (ZHANG; GUPTA; JHA, 2005), Zhang et al. proposed an algorithm for the majority-based logic synthesis of 3-input functions. In short, the proposed algorithm searches for patterns in Karnaugh map such that the target function can be represented by a majority function of three patterns. This method relies on the fact that any 3-input function can be implemented with at most 4 majority gates and two logic levels. As a consequence, this approach is not easily extensible to functions of more than 3 inputs because more logic levels would be required.

In (KONG; SHANG; LU, 2010), Kong et al., in turn, proposed a synthesis method to obtain optimal majority-based implementations of functions with at most 3 inputs. The algorithm starts from a set of 40 primitive functions that are implemented with at most one majority gate. To obtain the remaining 3-input functions, one majority gate is added. For each input of this additional gate is assigned one of the 40 primitive functions. As in (ZHANG; GUPTA; JHA, 2005), this method also relies on the fact that any 3-input function can be implemented with

two logic levels. This approach gives optimal results for 3-input functions. However, it is not suitable for a larger number of inputs due to the huge number of required combinations. When the number of inputs changes from 3 to 4, the number of primitive functions rises from 40 to 90. Moreover, the maximum number of majority gates required to implement the Boolean function changes from 4 to 9. That is the case when implementing the 4-input Exclusive-OR (XOR4) function.

In (WANG et al., 2013), Wang et al. generated the majority-based implementations of all functions with at most 4 inputs. The proposed method applies a bottom-up strategy in which simpler functions are combined to obtain more complex ones. The results of such method are optimal regarding number of majority gates. However, their method cannot be easily extended to consider different gates rather than majority gates.

In (MOMENZADEH et al., 2005), Momenzadeh et al. proposed a QCA implementation for an AND-OR-Inverter (AOI) gate. Even though the AOI is composed of 2 majority gates, its final area is smaller than the area of 2 majority gates due to the physical layout optimization. This particular implementation of AOI gate also includes negated inputs, which can be advantageous regarding area reduction when compared to the circuitry implemented using only majority gates and inverters.

The previous methods aim to generate all functions with a maximum number of inputs. Nevertheless, since the number of functions increases exponentially with the number of inputs (2^{2^n}), these methods are not able to synthesize functions with five or more inputs. In this sense, a different strategy to improve circuit design is to consider different gates as basic building blocks rather than only majority gates.

In this sense, it is interesting a methodology to evaluate how cell libraries with different composition impact the area of QCA designs and a novel cell library that exploits both majority and AOI gates as basic elements. Previous works consider either majority or AOI functions but not both. Even though the proposed library contains functions with at most 3 inputs, results have shown significant circuit area reduction when compared to related already published data.

6.2.2 Properties of the Majority Function

The majority function is a three input function represented by the following expression:

$$maj(a, b, c) = a \cdot b + a \cdot c + b \cdot c \tag{6.1}$$

The majority has 2 important properties. The first is the complete symmetry between its inputs. The second property is the self-duality, allowing propagate the inverters without changing the logic. Also, the majority function is symmetrical, i.e., any input permutation gives the same function.

Moreover, the majority function is not functionally complete, because to implement functions with negated literals the NOT operator (INV gate) must also be considered.

Typically, Boolean functions are represented as expressions using AND, OR and NOT as operations. Any expression of such form is trivially transformed into an equivalent majority expression (i.e., using the majority function as basic element) using the following relationships:

$$\begin{aligned} AND(a,b) &= maj(a,b,0) \\ OR(a,b) &= maj(a,b,1) \end{aligned} \tag{6.2}$$

It is often desirable to represent functions using as few operators as needed. Despite its simplicity, the trivial transformation method typically uses more operators than necessary. As example, consider the following function:

$$f(a,b,c) = a \cdot (b \cdot c + \overline{b} \cdot \overline{c} \tag{6.3}$$

One can transform all ANDs and ORs in the circuit into majority gates, but this strategy will not harness the majority functionality in the circuit. Since the trivial transformation does not guarantee optimal results, different methods have been proposed to obtain better implementations (ZHANG et al., 2004; ZHANG; GUPTA; JHA, 2005; KONG; SHANG; LU, 2010; WANG et al., 2013).

The standard design flows usually divide the target circuit into small functional blocks that have known implementation and design cost (possibly more than one cost). For this reason, works discussing majority-based logic synthesis have focused on the synthesis of simple functions. This way, traditional circuit synthesis flows can be applied to majority-based synthesis.

6.2.3 AOI Based Logic Synthesis

A natural way to extend the set of basic building blocks is to consider structures comprising more than one majority gate as a basic building block. For instance, the implementation of two majority gates connected in series can represent and AND-OR-Inverter gate, as expressed in

the following:

$$AOI(a,b,c,d,e) = d \cdot e + (d+e) \cdot (a \cdot (b+c) + b \cdot c) \tag{6.4}$$

However, the utilization of different basic building blocks is only useful if such blocks can be implemented in an optimized way when compared to the simple association of basic gates. For example, in CMOS design, complex gates can be implemented without being an association of NAND, NOR, and INV gates.

In (MOMENZADEH et al., 2005), Momenzadeh et al., proposed an optimized implementation for two series-connected majority gates, hereafter referred to as AOI gate. Due to the performed layout optimizations, the final area of the proposed is 1.77 larger than the area of a majority gate, instead of two times larger. Another interesting property of the gate proposed in (MOMENZADEH et al., 2005) is that the inputs of the second majority gate are inverted, changing the logic behavior from Equation (6.4) to Equation (6.5):

$$AOI(a,b,c,d,e) = d \cdot e + (d+e) \cdot (a \cdot (b+c) + b \cdot c) \tag{6.5}$$

The presence of negated inputs, without the explicit existence of an inverter gate, is an attractive feature due to the cost of adding inverters in QCA designs. The inverter area in QCA technology is 1.66 times greater than the majority gate area, which causes the inverter to be considered area expensive. Therefore, negated inputs with no area penalty have potential to reduce the design area. The trivial implementation of Equation (6.5), using majority gates and inverters, requires 2 majority gates and 2 inverters. Therefore, the normalized area of the straightforward implementation of Equation (6.5) is 5.32. This means that such an implementation is 3 times greater than the AOI gate proposed in (MOMENZADEH et al., 2005). It must be noticed that when the AOI gate is used in a circuit, the negated inputs change from the second to the first stage. Thus, Equation (6.5) is modified into Equation (6.6):

$$AOI(a,b,c,d,e) = d \cdot e + (d+e) \cdot (\overline{a} \cdot (b+\overline{c}) + b \cdot \overline{c}) \tag{6.6}$$

Using the AOI gate as only basic primitive (no single majority gates are used), Momenzadeh et al. implemented the same set of 13 functions presented in (ZHANG et al., 2004). Results have indicated that implementing circuits using only AOI gates can be advantageous regarding area compared to circuit implementations using only majority gates and inverters (MOMENZADEH et al., 2005).

6.2.4 Library Creation Methodology

The main idea of previous works considering majority-based logic synthesis is to implement large circuits using smaller gates (ZHANG et al., 2004; ZHANG; GUPTA; JHA, 2005; KONG; SHANG; LU, 2010). In overall, all proposed methodologies consist of two main steps: (1) decomposing the circuit in n-feasible functions, and (2) determining the implementation of each n-feasible function. Existing methodologies simply use the well-known SIS tool (WANG et al., 2013) to perform circuit decomposition. Thus, there is no significant difference between the decomposition step used in each methodology besides different scripts that perform the task of decomposition. In the second step, some works obtain the implementations by synthesizing each n-feasible on-the-fly (ZHANG; GUPTA; JHA, 2005) whereas others pre-synthesize all n-feasible functions and then use a look-up table to determine the implementation of a given n-feasible function (ZHANG et al., 2004; KONG; SHANG; LU, 2010). In this sense, all previous methodologies determine a set of functions (library), and they differ from each other on the value used for n (3 or 4) as well as on the implementations used for each n-feasible function in the library.

From the previous discussion, it is clear that the main difference of existing majority-based synthesis methodologies is the implementation of the functions in the library. Therefore, to perform a fair comparison among different synthesis approaches, it is interesting to have a generic library generation methodology. Such methodology should be able to consider different values for n as well as diverse basic building blocks.

The algorithm to synthesize majority gates is presented in Algorithm 8. The method CREATE_INITIAL_FUNCTIONS (line 3) generates the set of all 0-cost functions. These functions are input variables in both directed and complementary forms as well as constants true and false, which are in optimal form. The method ASSOCIATE (line 9-21) represents the partial order in FC. The method to synthesize majority gates is as follows (line 11-15). GET_INDECES_MAJ (line 12) is implemented according to the cost function. The cost function is chosen as the number of majority gates although any other cost function, such as logic depth and number of interconnections, could be addressed similarly. The term k-cost function is used hereafter to refer to a function with cost k, i.e., the implementation of f requires k majority gates.

Since the majority function is symmetrical, there is no need to consider different ordering for f_1, f_2 and f_3. In order to generate k-cost functions, all (k-1)-cost functions must be already known. When an implementation for a function f is first found with a cost c, the optimal

implementation cost for f is c. In other words, f is a c-cost function. The general composing rule is that when (C+1)-functions are generated, three functions f_1, f_2 and f_3 are combined such that the sum of their costs is C (line 14).

Algorithm 8 FC-EXACT-MAJ-AOI Algorithm

```
1:  function CREATE_ALL_FUNCTIONS (n, useMAJ, useAOI)
2:      i ← 1
3:      B[0] ← CREATE_INITIAL_FUNCTIONS ()
4:      while any function is not synthesized do
5:          B[i] ← ASSOCIATE (B, i, useAOI)
6:          i ← i + 1
7:      return B
8:
9:  function ASSOCIATE (B, i, useMAJ, useAOI)
10:     S ← ∅
11:     if useMAJ then
12:         indecesMaj ← GET_INDECES_MAJ (i)
13:         for each idx ∈ indecesMaj do
14:             ⟨i, j, k⟩ ← idx
15:             S ← S ∪ COMBINE (B[i], B[j], B[k], MAJ)
16:     if useAOI then
17:         indecesAOI ← GET_INDECES_AOI (i)
18:         for each idx ∈ indecesAOI do
19:             ⟨i, j, k, l, m⟩ ← idx
20:             S ← S ∪ COMBINE (B[i], B[j], B[k], B[l], B[m], MAJ)
21:     return S
```

6.2.4.1 Example using Logic Depth Approach

The generation of a majority-gate library comprising 2-input functions is illustrated in Figure 6.5. In the 0-depth bucket are allocated all variables in the positive and negative polarity and the constants. In the 1-depth bucket, all functions can be synthesized with just one majority gate. In the 2-depth bucket, the 'light gray' majority gates from the 1-depth $(\overline{a} \cdot \overline{b}, a \cdot b)$ are connected with the 0 constant in a majority gate to compose the $\overline{a \oplus b}$ function. In the same way, the dark gray majority gates from the 1-depth $(\overline{a} \cdot b, a \cdot \overline{b})$ are connected with the 0 constant in a majority gate to compose the $a \oplus b$ function. All 2-variable functions are covered in the three buckets. Considering all 16 possible functions of 2 inputs, 6 functions are in the 0-depth bucket, 8 functions are in the 1-depth bucket, and 2 functions are in the 2-depth bucket.

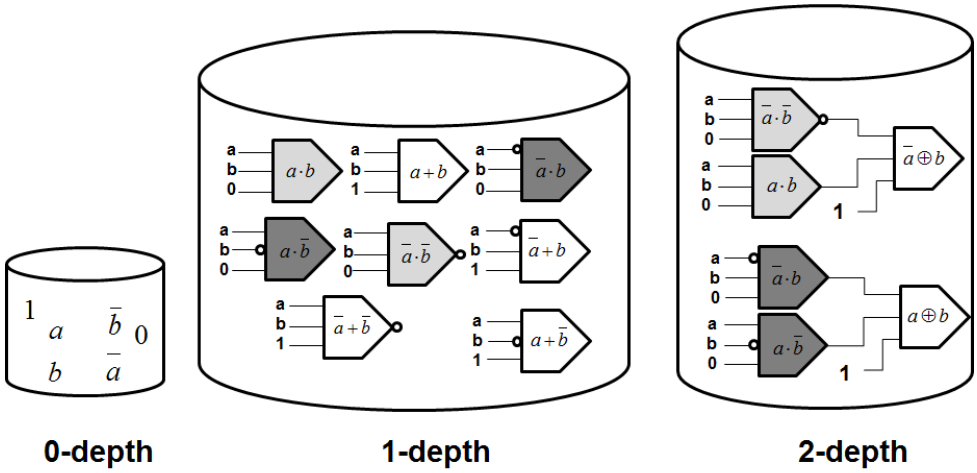


Figure 6.5 – Generation of all functions up to 2 variables using majority gates.

6.2.4.2 Synthesis using the AOI gate

The procedure to generate functions considering AOI gate, represented by Figure 6.6 and by Equation (6.6), is also described in Algorithm 8. To generate a function f, five functions are needed. However, since not all inputs are symmetric in the AOI gate, different input orders must be considered. This means that changing the order of inputs can change the gate functionality. As an example, if a and e are permuted, the resulting Boolean function is not logically equivalent to Equation (6.6). More specifically, there are two symmetric groups which are $[d, e]$ and $[a, c]$ and one anti-symmetric group that is $[b, \{a, c\}]$. As the equation Equation (6.6) has 4 AND operations, 4 OR operations and 4 negations, it is necessary 12 Boolean operations to synthesize an AOI gate (line 16-20).

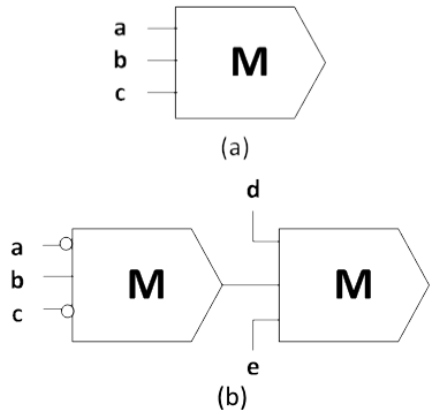


Figure 6.6 – Representation of logic gates used in this work. (a) majority gate. (b) AOI gate implemented using 2 majority gates.

The generation of a *mix* (MAJ+AOI) library comprising 2-input functions is illustrated in Figure 6.5. The main differences in the 1-bucket are almost all functions are implemented using

an AOI gate, except the AND2 and OR2 function. The 2-bucket uses an AOI gate instead of 2 MAJ gates to implement the XOR/XNOR function.

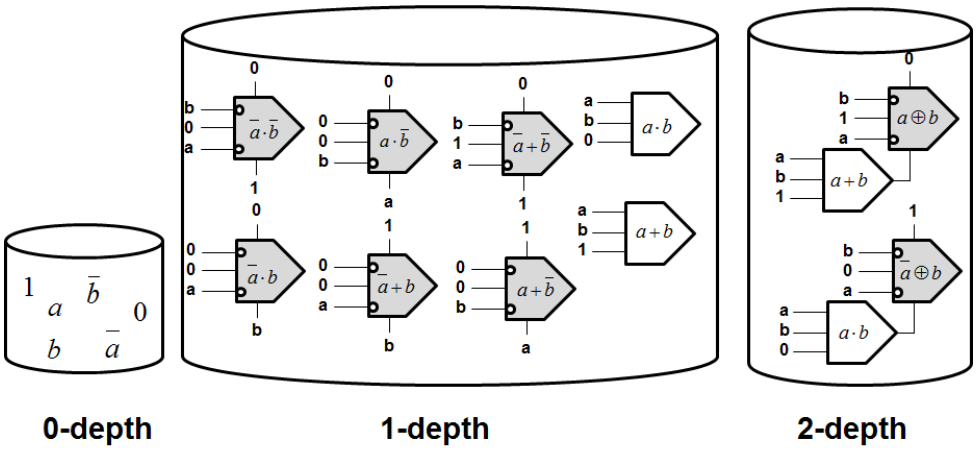


Figure 6.7 – Generation of all functions up to 2 variables using majority and AOI gates.

Once a library is generated, the functions in the library can be used in the synthesis of larger circuits. This is a common strategy also applied to traditional CMOS designs. That way, it is possible to investigate the impact of different libraries on digital design using QCA technology.

6.2.5 Circuit Synthesis Methodology

As already mentioned, a common step in existing majority-based circuit synthesis is to use SIS (SENTOVICH et al., 1992) to decompose the circuit in n-feasible functions. Even though the possible utilization of well-known tools is desirable for emerging technologies, some care must be taken when comparing the results. In short, even if all works rely on the same tool, the way that such tool is utilized can bias the results. This is an issue because improved results can be simply the consequence of better utilization of an already existing tool rather than the consequence of a novel idea, which is the main focus of the work.

Arguably, if all scripts used are provided, the adequate utilization of an existing tool can be considered part of the proposed circuit synthesis methodology. Even though providing the scripts used is somehow essential to allow some reproducibility of results, comparison with previous works should make clear what is the gain obtained due to better utilization of existing tools. Moreover, it should be considered the impact of using a better script on previous works. Another issue is the fact that there is no optimal script. In this sense, it is possible that if script A is used then, methodology 1 is better than methodology 2. However, when script B is used, methodology 2 could be better than methodology 1.

In this work, a straightforward flow is proposed to synthesize circuits using QCA technology. The aim of such flow is to reduce scripting bias that is present in related works (ZHANG et al., 2004; ZHANG; GUPTA; JHA, 2005; KONG; SHANG; LU, 2010; WANG et al., 2013; MARTINS et al., 2013). In this sense, it is not the goal of this work to obtain the best possible design implementation for each circuit. Instead, it aims to provide a circuit synthesis flow in which, for a given circuit, only the target library changes. This way, a fair comparison between existing methodologies can be performed.

The proposed flow consists of three steps:

1. The circuit is loaded into ABC tool (Berkeley Logic Synthesis and Verification Group, 2013) and logic optimizations are performed. For this purpose, the"resyn" command is executed three times. This command reduces both the logical depth and the number of nodes in the circuit.
2. The circuit is decomposed in n-feasible functions using the command "if –K n –a". This command performs area driven technology mapping targeting FPGAs using LUTs of size n. From such an implementation it is possible to extract how many times each n-feasible function is instantiated. The value of n is properly chosen accordingly to the maximum number of inputs in a cell library.
3. The QCA implementation cost of each n-feasible function considered in the circuit implementation is used to calculate the final circuit area.

6.2.6 Experimental Results

The libraries used in the experiments are described in Table 6.3. It is worth to mention that Momenzadeh et al. (MOMENZADEH et al., 2005) proposed a handmade library with 13 functions using AOI gates. Analyzing the implementations of these standard functions, the function $f = \overline{a} \cdot b \cdot c + a \cdot b \cdot \overline{c} + \overline{a} \cdot \overline{b} \cdot \overline{c}$ is implemented with 3 AOI gates. However, this function can be implemented with only 2 AOI gates, as found by the proposed algorithm. The implementations of this function are shown in Figure 6.9.

The first experiment is a comparison of the 13 standard functions implemented using AOI (MOMENZADEH et al., 2005) with the MAJ+INV+AOI (MIX_LIB) implementation. More than half functions have an area improvement since some AOIs in the implementation can be replaced by a majority gate. The results are presented in Table 6.4.

Table 6.3 – Libraries used in the experiments.

Lib. alias	Max. inputs	Basic gates	Related work
MAJ3_LIB	3	MAJ, INV	(KONG; SHANG; LU, 2010)
MAJ4_LIB	4	MAJ, INV	(WANG et al., 2013)
AOI_LIB	3	AOI	this work
MIX_LIB	3	AOI, MAJ, INV	this work

Table 6.4 – MAJ+AOI+INV versus AOI implementation of 13 standard functions (MOMENZADEH et al., 2005).

Functions	Number of AOI (MOMENZADEH et al., 2005)		Number of MAJ+AOI+INV (proposed approach)				
	Number of AOI	Effective Area	Number of MAJ	Number of AOI	Number of INV	Effective Area	Area Improvement (%)
$f = a \cdot \overline{b} \cdot c$	1	1.771	0	1	0	1.771	0
$f = a \cdot b$	1	1.771	1	0	0	1	43.5
$f = \overline{a} \cdot (\overline{b} \oplus c)$	2	3.542	0	2	0	3.542	0
$f = \overline{a} \cdot b \cdot c + a \cdot \overline{b} \cdot \overline{b}$	3	5.313	1	2	0	4.542	14.5
$f = \overline{a} \cdot b + b \cdot \overline{c}$	1	1.771	0	1	0	1.771	0
$f = a \cdot \overline{b} + \overline{a} \cdot b \cdot c$	2	3.452	0	2	0	3.542	0
$f = ((b + \overline{a}) \cdot (\overline{c} + (b \cdot \overline{a})) \cdot (c + \overline{b} + a))$	3	5.313	0	2	0	3.542	33
$f = a$	1	1.771	1	0	0	1	43.5
$f = a \cdot (b + c) + b \cdot c$	1	1.771	1	0	0	1	43.5
$f = \overline{a} \cdot b +, \overline{b} \cdot c$	2	3.542	1	1	0	2.771	21.7
$f = \overline{a} \cdot b + b \cdot c + a \cdot \overline{b} \cdot \overline{c}$	3	5.313	1	2	0	4.542	14.5
$f = a \oplus b$	2	3.542	1	1	0	2.771	21.7
$f = a \oplus b \oplus c$	2	3.542	0	2	0	3.542	0
Average	1.85	3.27	0.54	1.15	0	2.58	21.1

Also, the applied approach synthesizes functions with 4-inputs minimally in logic depth. The results are not guaranteed minimal for majority gate count. The "number of majority gates" partial order approach was implemented to observe the difference. It was noted that the "number of majority gates" partial order approach differs from "logic depth" partial approach by only 34 functions of a total of 65536 functions. These functions have a reduction of 3 majority gates (from 11 to 8) at an increase of logic depth (from 3 to 4). One example of such case is the function 1669_{16}, with two different implementations depicted in Figure 6.8.

The results presented in are the consequence of the technology mapping process instead of LUT decomposition. In this sense, the library used in this paper (AOI_LIB) is different from the one presented in (MOMENZADEH et al., 2005), since our implementations are optimal.

Table 6.5 summarizes the results over 40 MCNC benchmarks (YANG, 1991). The MAJ3_LIB is used as the reference. Also, a positive number represents an increasing whereas a negative number accounts for a reduction in area or logic depth regarding the reference library.

Comparing the MAJ4_LIB to MAJ3_LIB, there is a slight average improvement of 5% in area and a significant average improvement of 15% in logic depth. Only a few benchmarks give a worse logic depth, and a considerable number give a worse area overhead. This is related to the

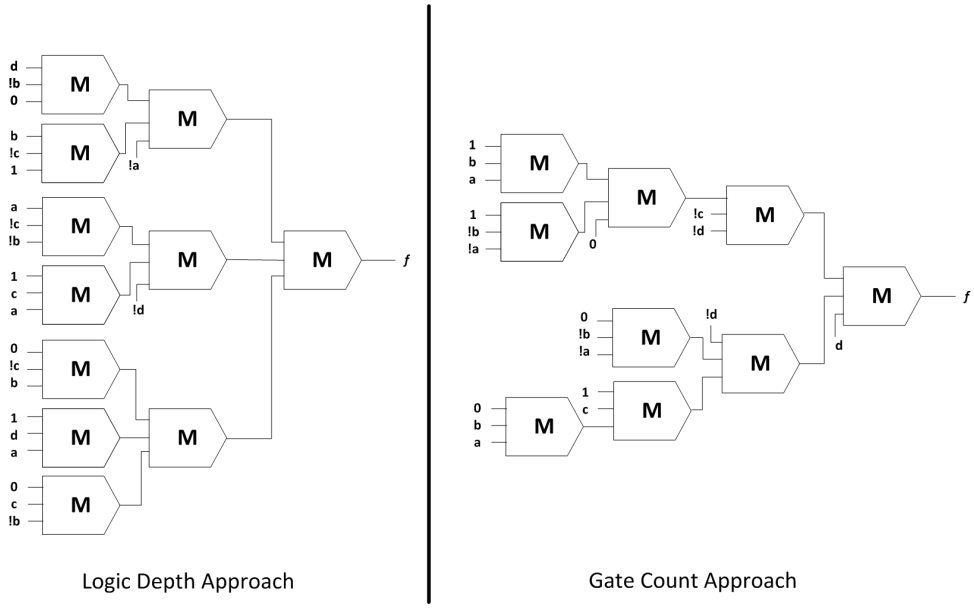


Figure 6.8 – Different implementations for the function 1669_{16}.

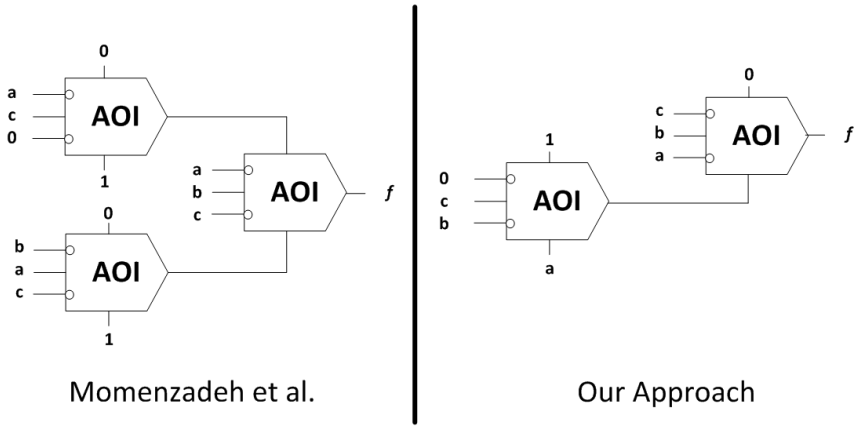


Figure 6.9 – Implementation of function $f = \overline{a} \cdot b \cdot c + a \cdot b \cdot \overline{c} + \overline{a} \cdot \overline{b} \cdot \overline{c}$ using AOI gates.

decomposition step which minimized the total number of functions but generated a significant number of 4-feasible functions compared to the K=3 decomposition.

Comparing the AOI_LIB to the MAJ3_LIB, there is a huge reduction (40%) of the area and a good reduction in the logic depth (14%). This reduction is explained by the fact of the computational power of the AOI gate, which contains 2 inverted gates, and eliminates the need for inverters in the circuit since an inverter can be built upon an AOI gate. The MIX_LIB is composed by implementations that use majority gates, inverters, and AOI gates, and reduce the circuit area even more than the AOI_LIB (47%), maintaining the logic depth achieved by the AOI_LIB. Interesting enough, the MIX_LIB provided an average area reduction of more than 42% compared to the MAJ4_LIB. This illustrates the importance of layout optimization in QCA design.

Table 6.5 – Results for MAJ/AOI synthesis over MCNC circuit benchmarks (MARTINS et al., 2014).

Benchmark	MAJ3_LIB Cost	MAJ3_LIB Depth	MAJ4_LIB Cost	MAJ4_LIB Depth	AOI_LIB Cost	AOI_LIB Depth	MIX_LIB Cost	MIX_LIB Depth
9symml	645.52	19	-24.9%	-15.8%	-44.6%	-10.5%	-51.1%	-10.5%
alu2	849.14	41	-3.0%	-22.0%	-43.9%	-17.1%	-50.3%	-17.1%
alu4	1565.65	42	-3.6%	-19.0%	-43.0%	-19.0%	-48.8%	-16.7%
apex6	1188.43	21	9.8%	-38.1%	-34.6%	-9.5%	-42.1%	-9.5%
apex7	324.94	17	-2.6%	-23.5%	-34.1%	-17.6%	-41.4%	-17.6%
c8	252.28	11	2.0%	-27.3%	-43.8%	-27.3%	-48.4%	-27.3%
cht	452.88	5	-46.9%	-20.0%	-56.6%	0.0%	-56.8%	0.0%
comp	218.62	13	-10.1%	7.7%	-40.9%	-15.4%	-46.2%	-15.4%
cordic	115.31	12	10.7%	-8.3%	-46.2%	-16.7%	-51.6%	-16.7%
count	293.27	20	2.8%	-30.0%	-39.0%	-35.0%	-51.6%	-35.0%
cu	109.64	10	-2.7%	-20.0%	-50.0%	-30.0%	-53.5%	-30.0%
dalu	2311.16	28	7.3%	-25.0%	-38.2%	-21.4%	-46.8%	-21.4%
des	8724.43	22	-7.6%	-9.1%	-41.2%	-9.1%	-48.5%	-9.1%
example2	456.91	11	10.7%	-18.2%	-29.1%	-18.2%	-37.0%	-18.2%
f51m	176.62	10	18.3%	-10.0%	-48.9%	-10.0%	-52.9%	-10.0%
frg1	244.94	20	4.5%	-5.0%	-41.4%	-15.0%	-49.9%	-15.0%
i10	3939.48	47	-2.4%	-21.3%	-39.3%	-27.7%	-45.2%	-27.7%
i2	534.87	15	4.5%	-20.0%	-51.7%	0.0%	-56.1%	0.0%
i3	285.94	7	-55.9%	-28.6%	-22.0%	0.0%	-34.9%	0.0%
i4	399.25	14	-38.4%	0.0%	-21.0%	-14.3%	-37.3%	-14.3%
i5	251.99	14	-5.7%	14.3%	-12.9%	-28.6%	-15.6%	-28.6%
i6	988.78	6	3.7%	-16.7%	-43.4%	0.0%	-54.5%	0.0%
i7	1558.65	6	-37.9%	-16.7%	-47.3%	0.0%	-50.9%	0.0%
i8	2259.48	19	-0.6%	-31.6%	-37.8%	-10.5%	-46.2%	-10.5%
i9	1573.62	15	-10.8%	-20.0%	-48.5%	0.0%	-52.9%	0.0%
k2	2739.08	20	5.1%	-5.0%	-34.5%	-15.0%	-44.7%	-15.0%
lal	165.63	10	12.7%	-20.0%	-43.4%	-10.0%	-48.5%	-10.0%
my	170.62	17	0.0%	0.0%	-33.6%	0.0%	-48.0%	0.0%
pair	3226.25	24	-7.5%	-8.3%	-41.1%	-12.5%	-49.2%	-12.5%
pcle	96.32	10	24.2%	-20.0%	-32.0%	-10.0%	-44.0%	-10.0%
pcler8	141.64	10	-16.5%	10.0%	-25.0%	0.0%	-38.6%	0.0%
sct	129.30	8	2.6%	0.0%	-43.8%	-12.5%	-47.4%	-12.5%
term1	214.29	12	-7.8%	0.0%	-42.2%	-25.0%	-49.0%	-25.0%
ttt2	330.92	11	2.4%	-9.1%	-45.4%	-9.1%	-50.3%	-9.1%
unreg	203.62	7	-5.2%	-28.6%	-57.4%	-42.9%	-57.4%	-42.9%
vda	1380.02	16	0.6%	-6.3%	-36.6%	-12.5%	-45.9%	-12.5%
x1	627.86	14	-0.8%	-28.6%	-42.2%	-21.4%	-49.0%	-21.4%
x2	105.31	7	-7.6%	0.0%	-51.2%	-14.3%	-54.9%	-14.3%
x3	1272.75	22	-1.2%	-45.5%	-33.1%	-9.1%	-43.5%	-9.1%
x4	795.14	10	-12.4%	-10.0%	-45.5%	-20.0%	-51.0%	-20.0%
Average	730.66	12.93	-4.8%	-14.9%	-40.2%	-14.2%	-47.3%	-14.1%

6.3 Synthesis of Spin-diodes circuits

In the spin-diode technology, being 2-input NOR and Exclusive-OR as basic gates, current CMOS-driven logic synthesis tools and algorithms are not necessarily the most efficient ones. The spin-diode logic family presents particularities that compromise the quality of results for algorithms developed specifically for CMOS IC design. For instance, the AND2 gate implemented using spin-diodes requires three diodes, which is expensive, compared to an NOR2 gate, which requires only one diode. Therefore, the use of CMOS algorithms has deleterious effects, as these algorithms are not optimized to avoid AND and utilize XOR gates. In this sense a more detailed analysis of the logic in spin-diodes is necessary.

In this section, we propose a set of algorithms that enable the efficient synthesis of circuits using the spin-diode technology. This modification consists of specialized algorithms for the synthesis of functions using diodes that consider the characteristics of the spin-diode technology to generate efficient implementations. To evaluate the quality of these algorithms, we will use these algorithms to transform a CMOS library into a spin-diode library.

6.3.1 Challenges of Spin-Diode Technology

In the remainder of this section, the following notation is used. The INV operation can be represented as ! or $\overline{bar}$. The NOR, XNOR, and OR operators are $-$, $\ominus$, and $+$, respectively. Also, the representation of the INV/NOR/XNOR gates is depicted in Figure 6.10. The wired-OR is a simple junction of wires.

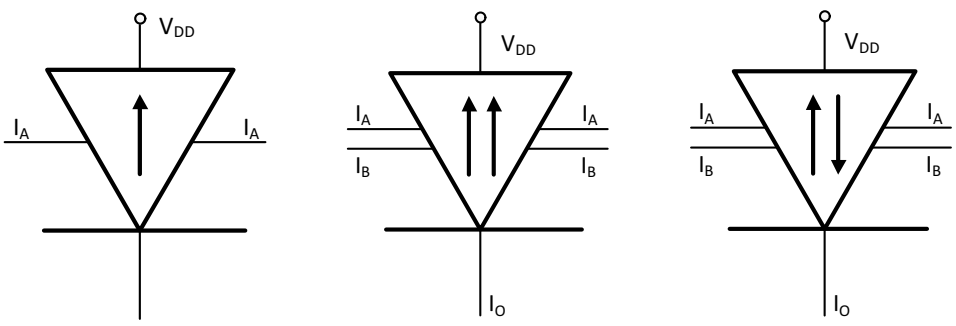


Figure 6.10 – Spin-diode schematic (a) INV (b) NOR (c) XNOR gate (FRIEDMAN et al., 2012b).

6.3.1.1 Use of Wired-OR Gate

As discussed earlier, OR operations can be performed by simply connecting two wires. However, when this operation is performed, the input signals are lost. If a wired-OR is used incorrectly, circuit operation can be compromised. As an example, an alternative implementation of the AND2 gate is considered. The AND2 function can be written as:

$$((A + B) \ominus A) \ominus B \tag{6.7}$$

Transforming Equation (6.7) directly into an equivalent diode network, the circuit shown in Figure 6.11 (a) is obtained. One possible method for improving this implementation is to replace the diode-OR with a wired-OR, aiming to obtain the circuit presented in Figure 6.11(b), which utilizes one less diode than the implementation shown in Figure 6.11(a). However, when a wired-OR is used, the actual implementation becomes as shown in Figure 6.11(c). As can be seen, the values of both A and B inputs are replaced by the value of $A + B$. Hence, the function implemented is:

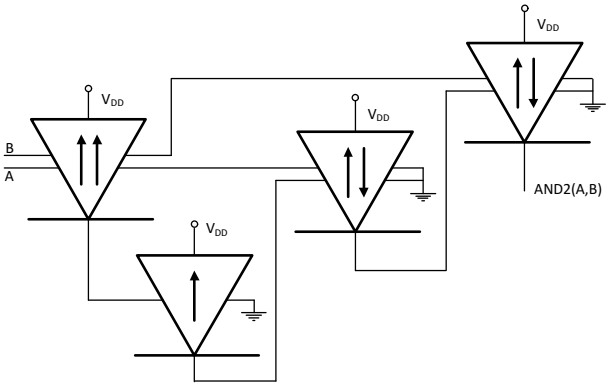
$$\begin{aligned} A \cdot B &= ((A + B) \ominus (A + B)) \ominus (A + B) \\ &= 1 \ominus (A + B) \\ &= A + B \end{aligned} \tag{6.8}$$

which is not the desired function. In some cases, the utilization of the wired-OR does not cause a gate malfunction but can have an adverse impact on other gates. Though this gate does implement the correct function, input 'A' is replaced with $A + \overline{B}$ throughout the circuit. Consequently, if A is also used in another gate, this gate receives the value of $A + \overline{B}$ instead.

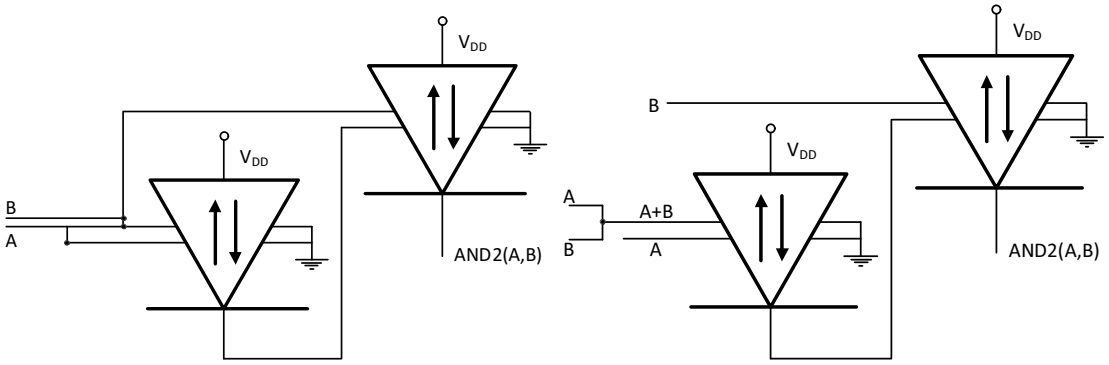
To ensure correct operation, a wired-OR is not used when: (1) any of the signals has fan-out greater than one or (2) when any of the signals is a primary gate input.

6.3.2 Implementation of Unate Functions

There are two interesting behaviors in unate functions. These behaviors are discussed in this section.



(a)



(b) (c)

Figure 6.11 – Second implementation of an AND2 function using Eq. 1 (a) with diode-OR (b) replacing the diode-OR by a wired-OR (c) the final result using wired-OR.

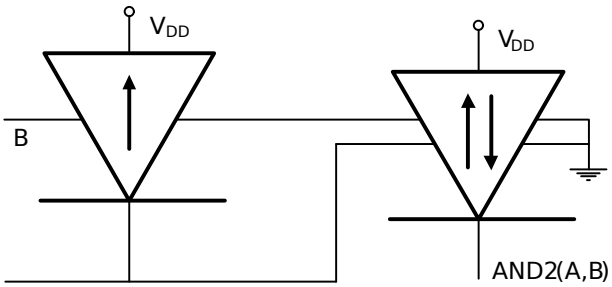


Figure 6.12 – Third implementation of an AND2 gate, considering fanin 1 for the input A.

6.3.2.1 XNOR Gates in Unate Functions

An interesting feature of spin-diode technology is the XNOR primitive gate. Since the XNOR is binate, the utilization of this gate in the optimal implementation of unate functions is counterintuitive. In order to explain this interesting behavior, consider the implementation (using spin-diodes) of the material implication function as:

$$IMP(A, B) = (A \ominus B + \overline{A}) \tag{6.9}$$

If we expand Equation (6.9) in order to achieve a SOP:

$$\begin{aligned} IMP(A,B) &= (A \cdot B + \overline{A} \cdot \overline{B}) + \overline{A} \\ &= A \cdot B + \overline{A} \\ &= B + \overline{A} \end{aligned} \tag{6.10}$$

Notice that the 'A' literal from the XNOR function, represented using a sum of products, is eliminated from the final expression, because of the absorption law applied in second to the last step.

6.3.2.2 Logic Sharing in Read-Once Functions

The read-once functions (LEE; WANG, 2007) are very important in circuits, as they have unique characteristics that play a central role in technology mapping and testability (KEUTZER, 1988). A read-once (RO) form is a factored form (using AND, OR, INV) where each variable appears at most once. Therefore, a Boolean function is an RO function if it can be represented by an RO form. For instance, the function $F(A,B,C,D) = A \cdot B + C\dot{D}$ is a read-once function because each variable of F in the factored form appears once. However, the function $G(A,B,C) = A \cdot (B + C) + B \cdot C$ is **not** a read-once function, as the variables B and C appear twice.

The read-once definition is based on Boolean basic operations (AND, OR, INV). As the spin-diodes have different basic operations (XNOR, NOR, OR, INV), the classical definition of read-once forms does not hold for spin-diode factored forms. CMOS factoring algorithms minimize literals in the expression. This is equivalent to reducing the number of operators. For the spin-diode technology, the relation is not direct. As the wired-OR has no implementation cost, reducing the number of spin operators does not imply reducing the number of devices. Moreover, factoring considers direct and negated variables with the same cost, whereas the inverter on spin-diode translates directly into a device.

As read-once forms are minimal, there is no possibility for logic sharing in CMOS. In spin-diode logic, there are read-once functions in which logic sharing occurs. For example, the OR3 function can be implemented as shown in Figure 6.13. Notice that the result of NOR(C, B) is used in both the INV and XNOR diodes, exemplifying the logic sharing.

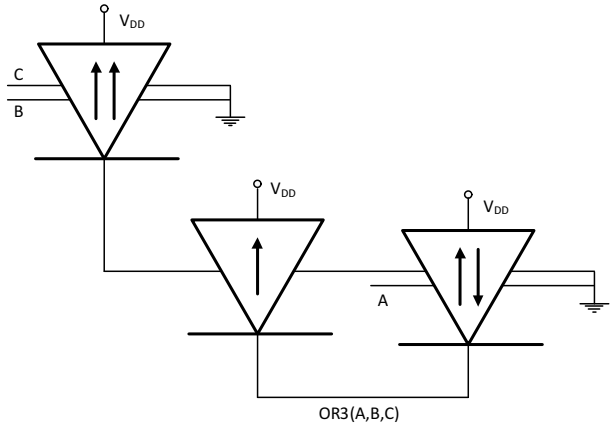


Figure 6.13 – Implementation of OR3 function using logic sharing.

6.3.3 Naive Implementation from Factored Forms

A common approach to design circuits using emerging technologies is to adapt established algorithms for CMOS technology. In the context of digital circuits using spin-diodes, a factorized CMOS expression, written using AND/OR and INV as basic operators can be directly transformed into an equivalent diode network. However, due to the very different primitive gates, achieving good results is not always possible. To illustrate this problem, consider the AOI22 function. The minimal factored form of AOI22 is:

$$AOI22(A, B, C, D) = (\overline{A} + \overline{B}) \cdot (\overline{C} + \overline{D}) \tag{6.11}$$

The direct transformation utilizes four inverters, one AND2 gate and a wired-OR. In this particular case, a two diode implementation of the AND2 gate (Figure 6.12) can be used and six diodes are required. This implementation has two more diodes than the minimal implementation ($(B - C) + (A - C) + (A - D) + (B - D)$). Another example is the majority gate, which is represented by the minimal factored form as:

$$MAJ(A, B, C) = A \cdot (B + C) + B \cdot C \tag{6.12}$$

To compute $B + C$ a diode-OR is required. To perform $A \cdot (B + C)$, a 2-diode-AND (Figure 6.12) can be used. The remaining $B \cdot C$ term needs a three-diode-AND ($\overline{B} - \overline{C}$). The final OR can be implemented using a wired-OR. Hence, a total of seven diodes is required to implement the majority gate. Clearly, the direct transformation is not efficient and more advanced methods should be investigated.

6.3.4 Fanout in Spin-Diode Technology

In a circuit, the same signal can be used by several different gates. In a traditional CMOS implementation, all receiving gates can be placed in “parallel” as shown in Figure 6.14(a). However, as the number of receiving gates increases the load capacitance of the driver gate also increases. This leads to a slower signal transition that can prejudice the circuit timing and power performances. Similarly, the number of transistors, as well as their size, controlled by the signal in a single receiving gate also impacts the output capacitance of the driver gate. CMOS circuits should be synthesized considering these factors.

If the spin-diode technology is targeted, the signal distribution is different. Since the spin-diode logic family is current-based, the same signal can drive as many gates as required without concern regarding the increased load capacitance (FRIEDMAN et al., 2012a). However, receiving gates cannot be placed in parallel because the input current for each gate is reduced, leads to a logical error (FRIEDMAN et al., 2012a). In the spin-diode logic family, the receiving gates should be connected as shown in Figure 6.14(b).

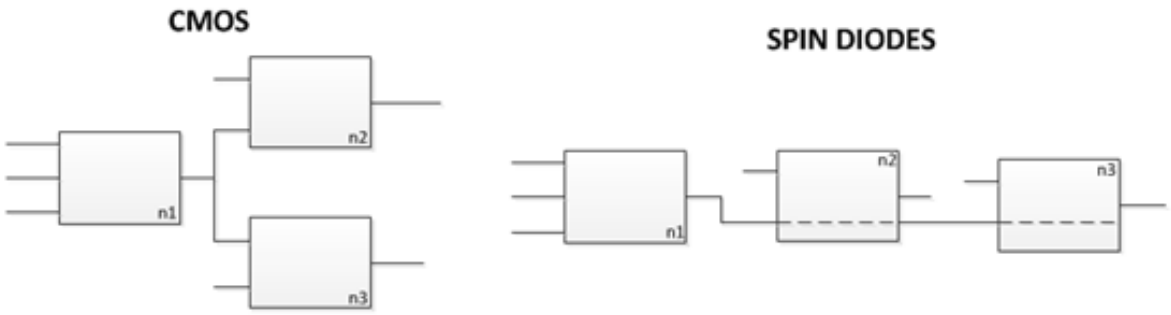


Figure 6.14 – Fanout in CMOS (a) and in spin-diode technology (b).

In short, there are two main differences between CMOS and spin-diodes regarding signal fan-out. Spin-diodes have a significant advantage in the fact that the delay of the gate is insensitive to the fan-out and, consequently, there is no limit to the number of gates a gate can drive. This fact should simplify logic synthesis. On the other hand, physical synthesis using spin-diodes tends to become more difficult because routing algorithms have less freedom.

6.3.5 Synthesis of Boolean Functions Using Spin-Diodes

As presented in Section 2.2.3, the basic logic gates using magnetoresistive spin-diodes, hereafter referred simply as diodes, are the XNOR/NOR/INV gates. The OR is a special case, where two wires are connected, avoiding the use of diodes. It is, therefore, important to develop an algorithm that can synthesize Boolean functions using the basic gates of the spin-diode technology, and use these synthesized functions as cells in a standard cell design flow. To implement efficient algorithms able to synthesize spin-diode logic, we need to take advantage of the functions implemented by the primitive gates (INV, XOR2, NOR2, OR).

The following algorithms to synthesize Boolean functions into spin-diodes using FC will be proposed. The first one synthesizes all functions up to four inputs. This algorithm is then improved with a logic sharing algorithm. The second algorithm is a modification of the algorithm presented in (MARTINS et al., 2010), considering the spin-diode operators, being able to treat functions up to eight inputs. The third algorithm traverses an AND/OR expression tree, translating then in a spin-diode tree, considering the wired-OR and 2-diode AND.

6.3.5.1 Algorithm 1: Generate all functions up to 4-inputs Considering Logic Sharing

The algorithm FC-SPIN (MARTINS et al., 2013) can generate all minimal fanout-free functions up to four variables, considering the set of operators INV, NOR, XNOR,OR, being the OR optional.

Algorithm 9 shows the pseudo code for FC-SPIN. CREATE_ALL_FUNCTIONS is the main method of the algorithm and will generate all minimum implementations up to n inputs. B is a list of sets of bonded-pairs, which will contain all implementations, separated by the number of diodes. First, all possible implementations containing zero diodes (i.e., variables in direct form) will be stored in a set (line 3). The while loop (line 5-7) generates all possible implementations in a crescent order, through the ASSOCIATE method. The main idea is to combine previous implementations to generate new ones, with cost i. The procedure ASSOCIATE has the task of creating all function with i diodes. This is performed inserting an inverter on top of all functions implemented with $i - 1$ diodes (line 11), and combining two functions, using NOR and XNOR diodes or the wired-OR (line 14-16), using the COMBINE method. The COMBINE method simply does a Cartesian product between two sets, using the operator defined (OR, NOR, XNOR), and if the resulting function is already synthesized, the implementation is discarded. Moreover, in the traditional CMOS approach, the combination of two functions also increases the cost of the resulting function, regardless of how the functions are combined. When using spin-diode

logic, this observation no longer holds because the wired-OR operator has cost zero. Also, there are restrictions on the utilization of the OR operator, as discussed in Section 6.3.1.1. Indeed, the partial order chosen is the number of diodes implemented (differently from the number of literals presented in Section 5.1.

Algorithm 9 FC-SPIN Algorithm

```
1:  function CREATE_ALL_FUNCTIONS (n, useOR)
2:      B ← ∅
3:      B[0] ← CREATE_INITIAL_FUNCTIONS (n)
4:      i ← 1
5:      while any function is not synthesized do
6:          B[i] ← ASSOCIATE (B, i, useOR))
7:          i ← i + 1
8:      return B
9:
10: function ASSOCIATE (B, i, useOR)
11:     S ← CREATE_INV (B[i − 1])
12:     for k ← 0, (i/2) − 1 do
13:         l ← i − k − 1
14:         S ← S ∪ COMBINE (B[k], B[l], NOR)
15:         S ← S ∪ COMBINE (B[k], B[l], XNOR)
16:         if useOR = true then S ← S ∪ COMBINE (B[k + 1], B[l + 1], OR)
17:     return S
```

To further reduce the number of diodes implementing Boolean functions, a strategy applying logic sharing can be considered. The logic sharing in our approach is divided into two parts: the logic sharing algorithm for a spin-diode expression (called LSE) and the algorithm to generate all four-input functions with logic sharing (called FC-SPIN-LS). The LSE is a topological logic sharing, where the algorithm traverses the spin logic tree and looks for nodes that implement the same function. This is very similar to the extraction algorithm (MICHELI, 1994), where a common subexpression can be extracted from a Boolean network. The two main differences are that the extraction algorithm extracts common subexpressions from a Boolean network, where the LSE algorithm extracts subfunctions only from the tree. The second difference is that LSE algorithm can treat the wired-OR operation. For instance, suppose the following spin-diode expression:

$$F(A,B,C) = C \ominus ((A \ominus (C \ominus B)) + \overline{(C \ominus B)}))$$

$$G(B,C) = (C \ominus B)$$

$$F(A, C, G) = C \ominus ((A \ominus G) + \overline{G}))$$

In this case, the cost of F initially is 5 diodes. The extraction of G allows the reduction of one diode, allowing F to be implemented with 4 diodes. Suppose now the following case:

$$F(A, B, C) = C \ominus ((A \ominus \overline{(C \ominus B)}) + \overline{(C \ominus B)}))$$

$$G(B, C) = (C \ominus B)$$

$$F(A, C, G) = C \ominus ((A \ominus \overline{G}) + \overline{G}))$$

The reason that $G(B, C)$ can not be $\overline{(C \ominus B)}$ is because of destructive behavior of the wired-OR operator, as discussed in Section 6.3.1.1. The LSE algorithm does not perform logic sharing of the subfunctions directly connected to the wired-OR operator. In this case, F will be implemented with five diodes.

The algorithm FC-SPIN-LS is a modified version of the algorithm presented in (MARTINS et al., 2013), considering the LSE algorithm to count the number of diodes in a spin-diode expression. Due to the logic sharing, the partial order does not work as expected when the combinations are performed. For example, suppose the following pair of bonded-pairs to be combined using an OR operator.

$$F = A - (C - \overline{D})$$

$$G = B - (C - \overline{D})$$

$$H = F + G = (A - (C - \overline{D})) + (B - (C - \overline{D}))$$

Clearly, both F and G functions are implemented using 3 diodes (2 NORs and 1 INV), but H is implemented using only 4 diodes, due the logic sharing ($(C - \overline{D})$). The H function in can generate a cascade effect, where a higher cost (in number of diodes) I function could be created previously using the H function, needing to be reimplemented as well.

The modifications from FC-SPIN to FC-SPIN-LS are as follows. The first step is to treat the violations that occur due to the logic sharing, as presented earlier. The main idea is when a violation happens in the ASSOCIATE method, store all costs of these functions. Select the lowest cost c. The index i from CREATE_ALL_FUNCTIONS now must be $c + 1$, instead of $i + 1$. In other words, the algorithm will 'reset' to a position where it will redo all combinations using these newer functions. There are two possible cases when a violation occurs: The function does not exist yet, or it is already implemented with a higher number of diodes. If the function

does not exist, the implementation is inserted in the right set. If the function already exists, besides inserting the implementation in the right set, the old implementation must be deleted (each function must have only one implementation).

6.3.5.2 Algorithm 2: Factorize a Function using a Heuristic Approach

To synthesize functions with more than four variables, a heuristic approach is proposed. We implemented a modified idea of the Algorithm 3, adapting the heuristics for spin-diode logic. Algorithm 10 shows the pseudo code for the FC-SPIN-HEUR. The algorithm starts testing if the function is a constant or variable (line 2, 4). In this case, the solution is trivial. After this initial checking, the function is decomposed, and all cofactors are extracted (line 6). These cofactors are factored (line 7) by a recursive call to the main method (line 14). The CF variable (line 7) stores all implementations of the cofactors.

These cofactors (and their implementations) are now separated by the order concept. Two Boolean functions can be compared and classified according to their relative order, which can be: equal, larger (LG), smaller (SM) and not-comparable (NC). It is said that F_1 is larger (smaller) than F_2 when the on-set of F_1 is a superset (subset) of the on-set of F_2. Two functions are equal when they have equal on-set and off-set. They are not-comparable when the on-sets are not contained by each other. The rules of ASSOCIATE_COFACTORS and GENERATE_SOLUTIONS (line 8-9) are described in Table 6.6. These rules are originated from a Boolean analysis of associations that contribute to finding the target function f. The only difference between the ASSOCIATE_COFACTORS and GENERATE_SOLUTIONS besides the different set of rules are that new functions found in ASSOCIATE_COFACTORS are stored, whereas in GENERATE_SOLUTIONS are not, since this method aims only implementations of f. This algorithm can also take advantage from LSE algorithm, to reduce the number of diodes.

Table 6.6 – Association rules of FC-SPIN-HEUR.

RULES	
ASSOCIATE_COFACTORS	GENERATE_SOLUTIONS
SM + NC	
SM $\ominus$ NC	SM + SM
SM $\ominus$ SM	
LG - NC	LG - LG
NC + NC	
NC - NC	NC $\ominus$ NC
NC $\ominus$ NC	

Algorithm 10 FC-SPIN-HEUR Algorithm

```
1:  function FACTORIZE (f)
2:      if CONSTANT (f) then
3:          return SOLUTION_CONST
4:      if VARIABLE (f) then
5:          return SOLUTION_VAR
6:      COF ← GENERATE_ALL_COFACTORS (f)
7:      CF ← FACTORIZE_COFACTORS (COF)
8:      AF ← ASSOCIATE_COFACTORS (CF)
9:      SOL ← GENERATE_SOLUTIONS (AF)
10:     return SOL
11:
12: function FACTORIZE_COFACTORS (COF)
13:     for each g in COF do
14:         CF ← CF ∪ FACTORIZE (g)
15:     return CF
```

6.3.5.3 Algorithm 3: Transform a logic tree into a spin-diode network

The heuristic synthesis finds difficulties for functions with more than eight variables since the number of cofactors and combinations increase exponentially. In this sense, another approach is necessary to treat more than variables. The naive transformations as shown in Section 6.3.3 can be improved, considering a set of transformations. Table 6.7 indicates the set of rules developed for the LST (Logic tree to Spin tree Transformation) algorithm. The input of this algorithm is a binary logic tree (preferably a factorized one). This set of rules explores the following concepts: XOR symmetric and antisymmetric properties, DeMorgan Law and the 2-diode AND. The symmetric cases were omitted ($L \circ \overline{R}$). The notation is as follows. L and R represents the left and the right side of the logic tree operator, respectively. L_S and R_S represents the transformed part as the spin tree of L and R, respectively. When in the original tree L and/or R is inverted, L_S (R_S) has an inverter on top. $L_{S_{INV}}$ ($R_{S_{INV}}$) represents L_S (R_S) without the inverter.

Some rules cost zero diodes, as the $L_{S_{INV}} \ominus R$. This is because $L_{S_{INV}}$ is implemented with less one diode than L_S. Adding the cost of one diode through the XNOR operation, the transformation does not add diodes. Nevertheless, the number of diodes in each operation continues to be the sum of L_S and R_S. To count the total number of diodes, the LSE algorithm can be used.

Table 6.7 – Transformations applied to convert a logic tree into a spin tree.

Transformation rules	Logic tree	Spin tree	Transformation cost (# diodes)
AND	$L \cdot R$	$\overline{\overline{L_S} + \overline{R_S}}$	3
		$(\overline{L_S} + R_S) \ominus L_S$	2
	$\overline{L} \cdot R$	$\overline{R_S} - L_{S_{INV}}$	1
	$\overline{L} \cdot \overline{R}$	$L_{S_{INV}} - R_{S_{INV}}$	1
OR	$L + R$	$\overline{L_S - R_S}$	2
		$L_S + R_S$	0
	$\overline{L} + R$	$(R_S \ominus L_{S_{INV}}) + L_S$	1
		$L_S + R_S$	0
	$\overline{L} + \overline{R}$	$L_S + R_S$	0
XOR	$L \oplus R$	$\overline{(L_S \ominus R_S)}$	2
	$\overline{L} \oplus R$	$L_{S_{INV}} \ominus R_S$	0
	$\overline{L} \oplus \overline{R}$	$\overline{L_{S_{INV}} \ominus R_{S_{INV}}}$	0

6.3.6 Experimental Results

In this section, the quality of the proposed spin logic algorithms are evaluated (FC-SPIN, FC-SPIN-LS, FC-SPIN-HEUR, and LST), in comparison to the FC-SPIN algorithm of (MARTINS et al., 2013). Also, a technology mapping study using the standard cell methodology is performed. Benchmarks circuits are mapped to spin-diode libraries using a commercial tool. The platform used was an Intel Core i5 processor with 2GB main memory.

6.3.6.1 Algorithm Comparison

To analyze the proposed algorithms, we start evaluating the FC-SPIN algorithm, compared to the state-of-the-art ABC (Berkeley Logic Synthesis and Verification Group, 2013) and a commercial tool. The platform used was an Intel Core i5 processor with 2GB main memory.

The first step was the generation of an optimal fanout free spin-diode networks for all functions with up to four inputs, stored in a look-up table (LUT). To analyze the impact of considering the OR gate, two LUTs were generated. The first (FC-SPIN) considers the complete set of operators INV, NOR, XNOR, OR, taking into account the OR gate electrical characteristics in the network, as discussed in Section III, allowing OR gates only in internal nodes. The second (FC-SPIN w/o OR) is synthesized using a partial set of operators INV, NOR, XNOR.

The execution time to generate each LUT was about one minute. The file with the LUT

occupies less than 3 MB, and the LUT loaded in memory occupies less than 50 MB, so making it feasible in a logic synthesis tool. It is worth to mention that all 4-input functions were synthesized using at most nine spin-diodes when considering the OR gate, and ten gates when excluding the OR gate.

To analyze the synthesis of spin-diode networks using two logic synthesis tools, ABC, and a commercial tool was applied. These tools were used to perform the technology mapping with a set of 3982 representative 4-input functions, grouped into permutation equivalent classes (4-P set). The gates present in the library were: INV (1), AND (3), NAND2 (2), OR2 (2), NOR2 (1), XOR2 (2), and XNOR2 (1). The gate cost is defined as the required number of spin-diodes necessary to implement such gate, shown within the parenthesis. Since these tools cannot evaluate when it is safe to use the OR operator, this gate is provided only in the version NOR+INV to ensure a correct implementation. The scripts used in the ABC tool to perform the technology mapping used AIGs primarily with the choice approach or a supergate library approach, having area reduction as the main objective.

The results of the technology mapping performed in ABC and the commercial tool for each function is compared with the FC-SPIN algorithm, and a histogram is shown in Figure 6.15. Positive (negative) numbers indicate a worse (better) implementation based on spin-diode count, compared to the algorithm disallowing the wired-OR. It is noted that the FC-SPIN only guarantees a minimal fanout free network. In this sense, there are a few cases that these tools can exploit logical sharing to generate a network with fewer spin-diodes.

Another important consideration is the impact of the OR gate in the diode count, generating an implementation overhead in some functions with up to three diodes. On the other hand, the ABC and the commercial tools generate a considerable number of functions with more than five diodes compared to the optimal implementation, representing 1250 (31.39%) and 2022 (50.78%) functions, respectively.

An analysis of the overall results of the algorithms is shown in Figure 6.16, which shows the total diode count for the 4-P set. The FC-SPIN disallowing the wired-OR has an overhead of 4.63% against the version using the wired-OR. To have a fair comparison, the ABC and the commercial tools are compared to the FC-SPIN without the wired-OR, having an increase of 44.54% and 61.51% in diode count, respectively. These results demonstrate that logic synthesis tools are very far from optimality, mainly when considering technologies that do not have AND/OR/INV as basic gates. This considerable reduction allows generating very compact gates that will reduce the final area of circuits.

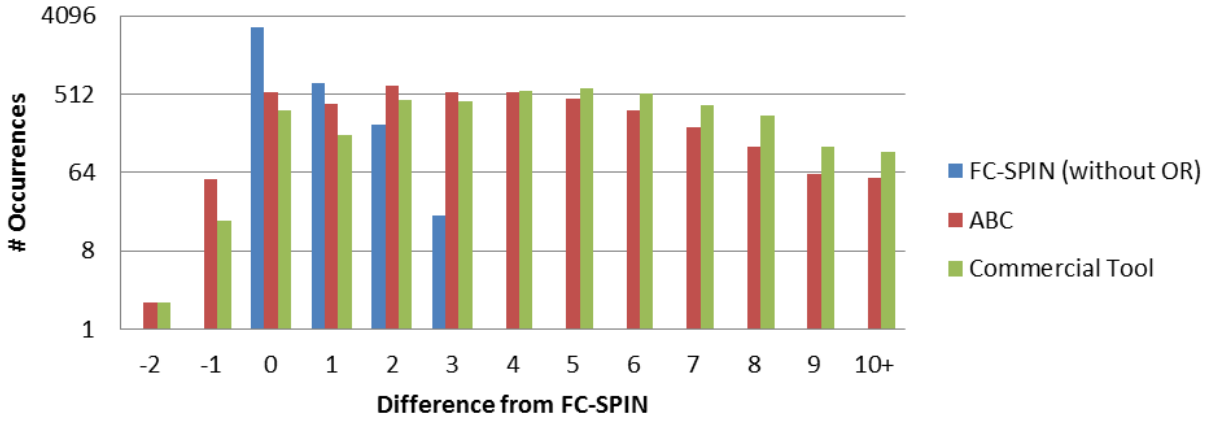


Figure 6.15 – Histogram of the differences from each algorithm compared to FC-SPIN using the wired-OR (MARTINS et al., 2013).

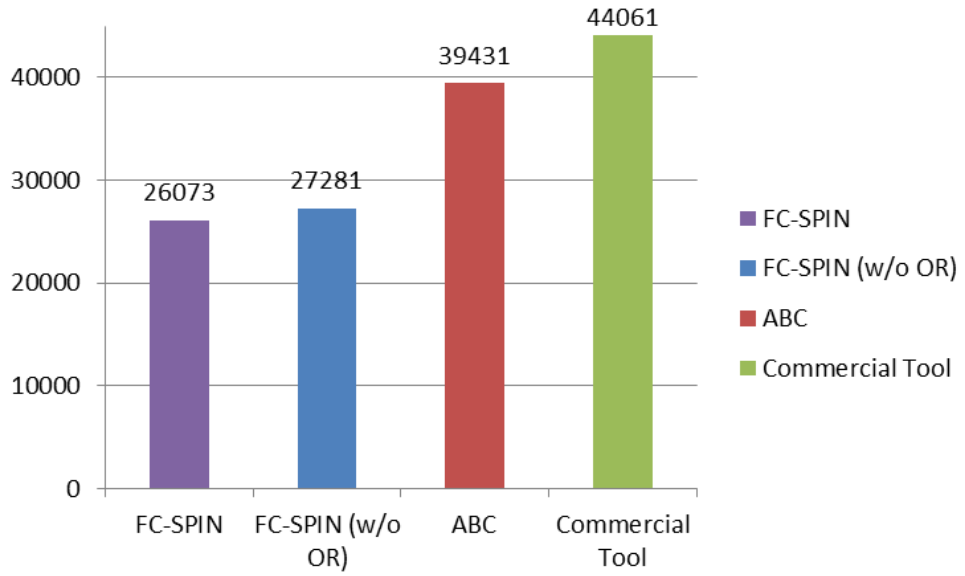


Figure 6.16 – Overall spin-diode count for the 4-P set (MARTINS et al., 2013).

After a detailed analysis of the FC-SPIN, two analyses are made: the improvement of logic sharing and the quality of the heuristics. The benchmarks used for both analyses are the set of all functions of up to four inputs. For the evaluation of the transformation rules, two factorized forms were used: all functions minimally factored up to four inputs, with or without the XOR operator (MARTINS; RIBAS; REIS, 2012). Additionally, the naive implementations were also synthesized using both sets of factorized forms. Also, synthesis with ABC (Berkeley Logic Synthesis and Verification Group, 2013) and a commercial tool were also included, for all functions of up to 4 inputs, with the same configuration and scripts used in (MARTINS et al., 2013). Figure 6.17 presents the results. The red bar represents the reference algorithm; the green bars represent the proposed algorithms, the orange bars the naive transformations and the blue bars, logic synthesis tools.

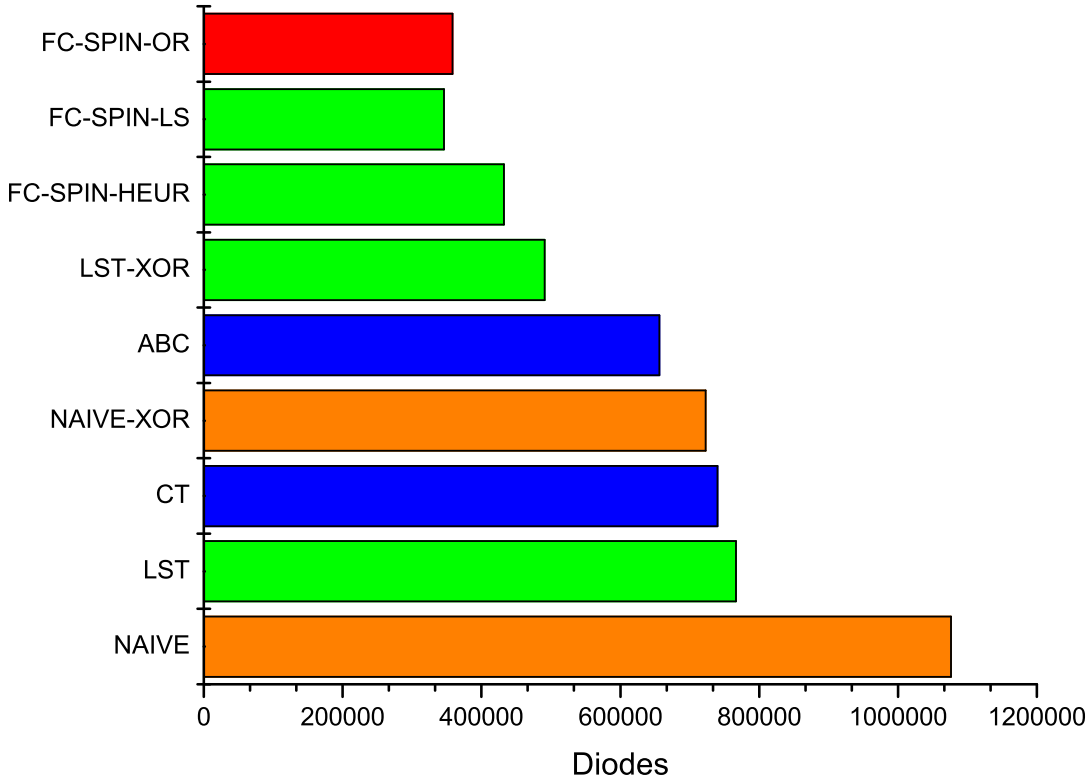


Figure 6.17 – Histogram of the differences from each algorithm compared to FC-SPIN (MARTINS et al., 2015b).

As expected, the FC-SPIN-LS can further reduce the number of devices, compared to FC-SPIN using the OR with a reduction of 3.3% in the total count. The set of improved functions represents 18.0% of all functions synthesized. Each function of this set had an average improvement of 16.4%, with a reduction of one or two diodes, due to logic sharing. The heuristic algorithm FC-SPIN-HEUR increased the number of the diodes in almost 21% compared to FC-SPIN. However, this algorithm can scale to eight variables, whereas both FC-SPIN and FC-SPIN-LS are not.

For more than eight variables, it is necessary to use the LST algorithm. This algorithm is heavily dependent on the input (a binary tree). Therefore, factored forms that included the XOR operation yield better results, as the logic trees tend to have fewer operators. This is confirmed by the results of LST using a factorized tree with and without XOR, increasing by 37.1% and 114.0%, respectively, when compared to FC-SPIN. The naive transformation of the factorized tree consists of directly translating the AND, OR, and XOR operators. The AND translation can be a two or three diode implementation (discussed in Section 6.3.3). The OR translation can be a wired-OR or a two diode implementation(NOR+INV). Finally, the XOR translation is a two diode implementation (XNOR+INV), and the INV translation is a one diode implementation. This transformation leads to bad results, increasing the number of diodes using the factored

forms with and without XOR, by 101.8% and 200.6%, respectively. The increase in the number of diodes compared to LST with and without XOR is of 47.2% and 40.4%, respectively.

The execution time to synthesize all four input functions in FC-SPIN is 2 minutes, in FC-SPIN-LS is 9 hours (due to the considerable number of rollbacks), in FC-SPIN-HEUR is 20 minutes, and in LST is less than 1 second. These results demonstrate that FC-SPIN-LS can reduce the number of the diodes. The heuristic algorithms presented, FC-SPIN-HEUR and LST, made a reasonable trade-off between quality and scalability. These algorithms will be applied in a standard-cell flow.

6.3.6.2 Examples of Synthesized Logic Gates

One possible advantage of spin-diodes over traditional CMOS is the likely reduction of the number of devices required to implement a Boolean function. Such a reduction is expected due to both the simple XNOR gate and the wired-OR. Additionally, in spin-diode logic, complementary devices are not needed. An example of the influence of the XNOR gate is a 3-input comparator circuit COMP3 ($F = A \cdot B \cdot C + \overline{A} \cdot \overline{B} \cdot \overline{C}$), the output of this gate is 1 *iff* all inputs have the same value. The implementation of the COMP3 gate utilizes five diodes (two XNORs, two inverters, and one NOR), as illustrated in Figure 6.18.

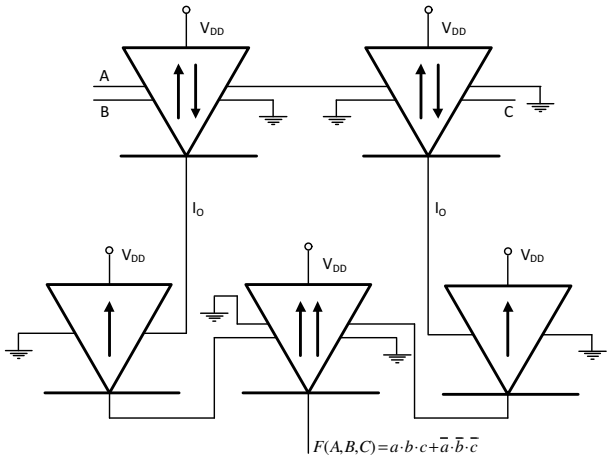


Figure 6.18 – Spin-diode logic circuit for $COMP3 F = A \cdot B \cdot C + \overline{A} \cdot \overline{B} \cdot \overline{C}$.

An example of the importance of the wired-OR is the implementation of the AOI22 function ($F(A, B, C, D) = \overline{A \cdot B + C \cdot D}$), which schematic is illustrated in Figure 6.19. The OR wire is applied to the output of the four diodes. Without the wired-OR, the cost of AOI22 is seven diodes.

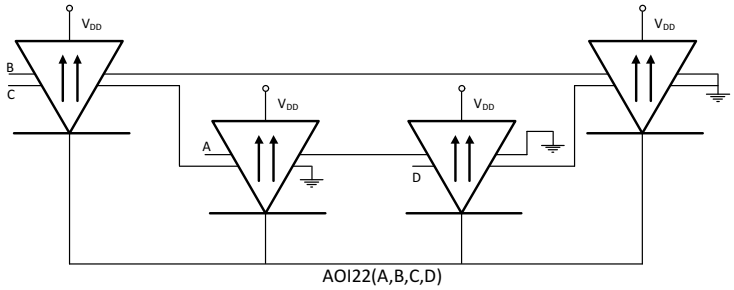
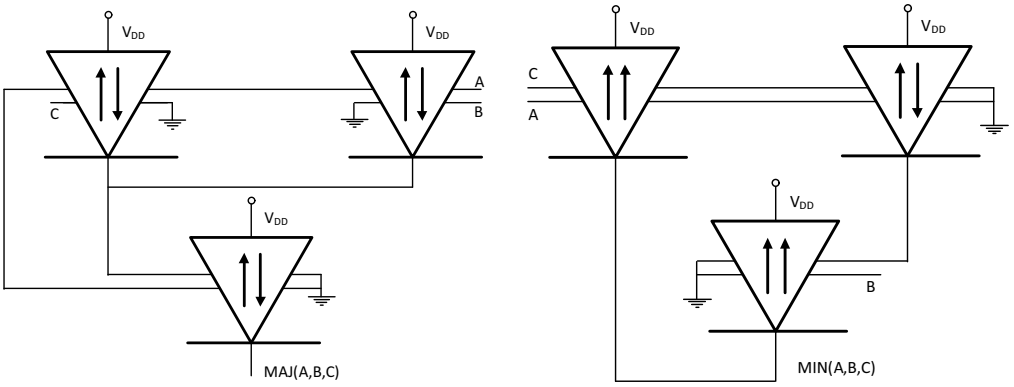


Figure 6.19 – Spin-diode logic circuit for $AOI22(A, B, C, D) = \overline{A \cdot B + C \cdot D}$.

Two other common gates in CMOS technology are the majority $MAJ(A, B, C) = A \cdot B + A \cdot C + B \cdot C$ and minority $MIN(A, B, C) = \overline{A} \cdot \overline{B} + \overline{A} \cdot \overline{C} + \overline{B} \cdot \overline{C}$ voters. Each of these gates is implemented using 3 diodes. Figure 6.20 shows the schematics for the majority and minority voters.



(a) (b)

Figure 6.20 – Spin-diode logic circuit for a majority voter (a) and minority voter (b).

The full-adder and half-adder are basic building blocks for the design of arithmetic circuits. The schematic of a half adder is depicted in Figure 6.21. In Figure 6.22, a full-adder implementation using five spin-diodes is presented. The multiplexer is another common element in the digital design, particularly in encoders and decoders. A 2-to-1 multiplexer can be designed using three diodes as shown in Figure 6.23. Both the full-adder and multiplexer presented in (FRIEDMAN et al., 2012b), utilize one more diode than the implementation proposed herein.

Sequential elements are also essential to digital design. An RS latch can be found in (FRIEDMAN et al., 2012b). A possible implementation for a D latch, using six diodes, is presented in Figure 6.24. Connecting two D latches in series, a positive edge, master-slave D flip-flop is obtained. It is interesting to notice that, similarly to CMOS technology, logic sharing

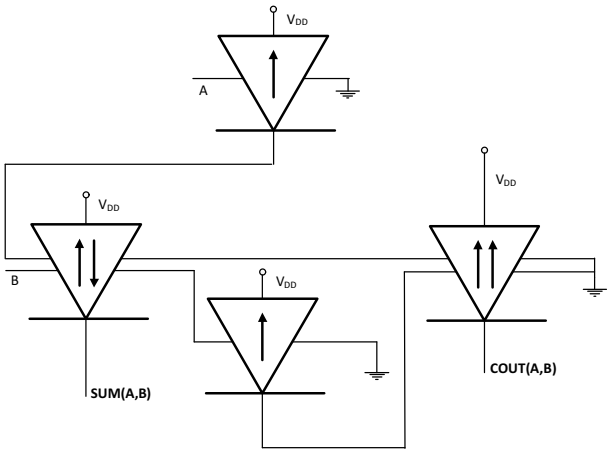


Figure 6.21 – Spin-diode logic circuit for a half adder.

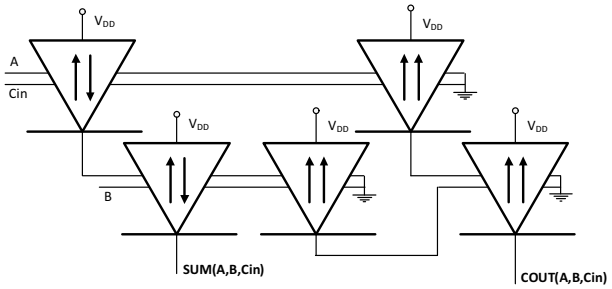


Figure 6.22 – Spin-diode logic circuit for a full adder.

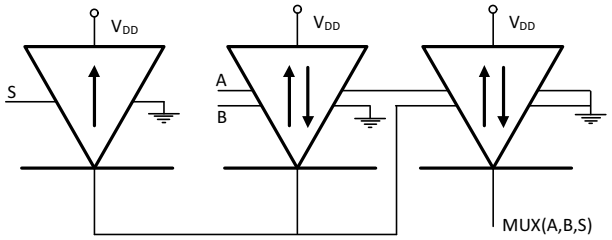


Figure 6.23 – Spin-diode logic circuit for a multiplexer.

can be used to reduce the number of diodes. Examples are both the half-adder (Figure 6.21) and the D latch (Figure 6.24).

6.3.6.3 Standard Cell Mapping Approach

One of the main design methodologies used in CMOS technology is the approach based on standard cells. In this work, a technology mapping study using the standard cell methodology is performed. Benchmarks circuits are mapped to spin-diodes libraries using a commercial tool.

It must be noticed that the results presented herein focus on the number of required devices to implement benchmark circuit. The area attribute of each gate is set to the number of diodes used. Though timing and power vary throughout a circuit, our analysis assumes uniform characteristics. This simplification is adopted due to the lack of a well-established electrical

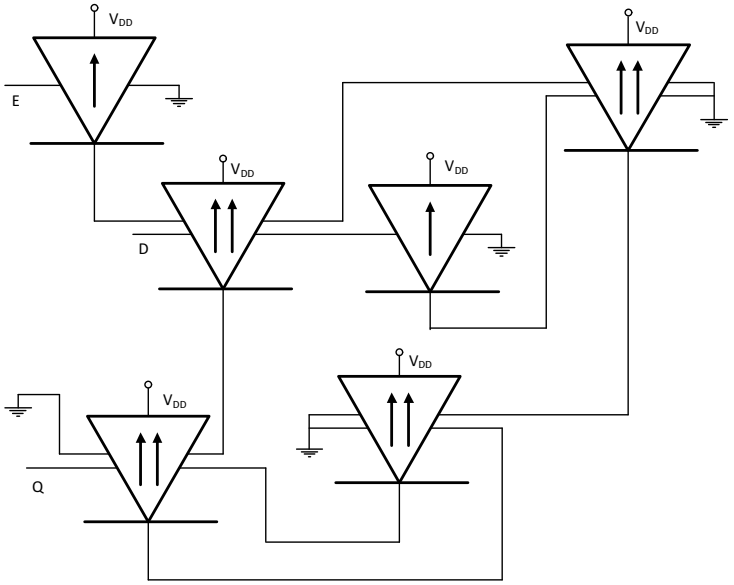


Figure 6.24 – Spin-diode logic circuit for a D latch.

model for the electrical characterization of spin-diodes gates. This is done by setting all entries in timing and power tables to the same constant value for all cells. The attribute that controls the maximum fan-out of a gate is set to a very high value. This is justifiable because a spin-diode should be able to drive as many gates as needed. Finally, since current-based logic is not affected by RC delay, wire delay is not considered.

Several sets of functions are considered. The MinLib library contains only the gates that require a single diode to be implemented (XNOR2, INV, and NOR2) and the OR2 gate implemented with two diodes. This library is referred to as MinLib. We also used a modified version of a 65 nm commercial library. For the combinational cells, we only selected the minimum drive strength cells. All functions were synthesized as follows. For cells having functions up to four inputs, the spin-diode implementation comes from a lookup table with the results of FC-SPIN-LS algorithm. Functions having from five to eight inputs are synthesized by the FC-SPIN-HEUR method. Finally, cells containing functions with nine or more inputs are translated using LST. We also removed all sequential cells, except for a D flip-flop (DFF), a DFF with asynchronous set and a DFF with asynchronous reset. The implementation used for the DFFs use a master-slave architecture. Moreover, we removed all tri-state and analog cells. With this reduced cell library, we also removed the timing and power information also and generated two versions, one with the original CMOS cell area (CMOSLib) and the other with the correspondent diode count (DiodeLib).

Technology mapping over IWLS 2005 benchmarks (ALBRECHT, 2005) are shown in Table 6.8. C. Area represents the combinational area of the benchmark; T. Area represents the total Area of the benchmark (in number of diodes). C. Inv represents the equivalent inverter (area

divided by the area of the smallest inverter) of the combinational part, and T. Inv represents the equivalent inverter area of the whole circuit. It can be seen that the DiodeLib reduces considerably the combinational area, compared to the MinLib. The reduction of the combinational area varies from 26.7% of stepper to 51.1% of pci. The total area is impacted by the area of a DFF (11 diodes), but there are old DFF designs based on NORs (TAUB, 1977), which can reduce the total number of spin-diodes, compared to a master-slave architecture. Nevertheless, the improvements in the total area range from 7% to 34.7%.

If we compare DiodeLib with CMOSLib in the combinational part, we can observe that the number of spin-diodes and equivalent inverters are almost the same for all circuits. Moreover, considering that the inverter is implemented with two transistors and diodes are smaller than transistors (i.e., a PN junction is smaller than an NPN junction), the area of a spin-diode circuit tends to be smaller than the area of a CMOS circuit.

Table 6.8 – Results for IWLS 2005 benchmarks (MARTINS et al., 2015a).

Benchmark	MinLib C. Area/T. Area	DiodeLib C. Area/T. Area	CMOSLib C. Inv/T. Inv.
aes_core	25792/32682	19411/26301	19328/22685
des_area	6004/6862	4212/5096	3983/4401
des_perf	88338/202842	68123/182627	71096.33/126880.3
pci	136/136	90/90	83.66/83.66
ss_pcm	456/1587	330/1461	334.33/885.33
stepper	147/472	116/441	121.66/280
tv80	7967/12634	5532/10199	5452.33/7726
usb_funct	15751/38241	10780/33296	11183.67/22146.67
usb_phy	563/1837	433/1707	403/1023.66

6.4 Synthesis of Memristor Implication-based Logic

Memristors and MTJ have been applied to perform logic using material implication as basic logic function (BORGHETTI et al., 2010; LEHTONEN; LAIHO, 2009; LEHTONEN; POIKONEN; LAIHO, 2010; LAIHO; LEHTONEN, 2010; KIM; SHIN; KANG, 2011; KVATINSKY et al., 2014; POIKONEN; LEHTONEN; LAIHO, 2012; TEODOROVIC; DAUTOVIC; MALBASA, 2013; MAHMOUDI et al., 2013a; MAHMOUDI et al., 2013b). As a consequence, several efforts have been made to propose new algorithms, or modifications of known algorithms, that can take into account the particularities of a given technology.

In (LEHTONEN; POIKONEN; LAIHO, 2010), it is shown that an arbitrary Boolean function of n inputs can be implemented using n+2 memristors. The implementation considered in (LEHTONEN; POIKONEN; LAIHO, 2010) requires that the function is written in a recursive form which implies that the computation is performed sequentially. Therefore, there is a direct link between the number of material implications performed and the computation time. For this reason, different approaches have been proposed to reduce the number of material implications in the recursive formulations (POIKONEN; LEHTONEN; LAIHO, 2012; TEODOROVIC; DAUTOVIC; MALBASA, 2013).

In this section, we present the concepts of the memristive implication logic, involving the concept of multi-input memristor implication. This concept is used to propose algorithms using FC to synthesize Boolean functions using memristive material implication synthesis. It is important to notice that the minimization of the implications increases the overall speed of the circuit, differently from the previous FC algorithms, which in general minimized an area cost (literals, threshold and majority gates, spin-diodes).

6.4.1 Implication Logic

The notation in this section is as follows. The material implication operation can be denoted by $\rightarrow$. Logical negation, conjunction, disjunction and exclusive disjunction are represented by $\neg$, $\wedge$, $\vee$ and $\oplus$ respectively. Alternatively, the notation $p \wedge 0 = p \rightarrow 0 = \neg p$ and $p1 = p$ are also used. A cube is a conjunction of literals, i.e., $p_1 \wedge p_2 \wedge p_4$. Any Boolean function with n inputs $f : B^n \rightarrow, B = 0, 1$, can be written as (TEODOROVIC; DAUTOVIC; MALBASA, 2013):

$$f = ((\ldots(\pi_l^{\alpha_l}) \vee \pi_{l-1})^{\alpha_{l-1}} \ldots \pi_2)^{\alpha_2} \vee \pi_1)^{\alpha_1} \tag{6.13}$$

Table 6.9 – Positive product terms for $n = 3$.

p_1	p_2	p_3	π_k
0	0	0	$\pi_8 = 1$
0	0	1	$\pi_7 = p_3$
0	1	0	$\pi_6 = p_2$
0	1	1	$\pi_5 = p_2 \rightarrow p_3$
1	0	0	$\pi_4 = p_1$
1	0	1	$\pi_3 = p_1 \rightarrow p_3$
1	1	0	$\pi_2 = p_1 \rightarrow p_2$
1	1	1	$\pi_1 = p_1 \rightarrow p_2 \rightarrow p_3$

where π_k is a positive product term of the form $\pi_k == \pi_{k1} \wedge \pi_{k2} \wedge \ldots \wedge \pi_{kh}$, each π_{kh} is in the support of f and $l = 2^n$. Given a lexicographical ordering, the positive product terms can be written as $\pi_l = p_1, \pi_{l-1} = p_2, \ldots \pi_1 = p_1 \wedge \ldots \wedge p_n$ where $p_1, p_2, \ldots p_n$ are the support of f. Table 6.9 gives all positive product terms for $n = 3(l = 8)$. The value of each α_k is given by:

$$\alpha_k = \begin{cases} f(\pi_1), k = 1 \\ \neg(f(\pi_k) \oplus f(\pi_{k-1})), k \neq 1 \end{cases} \tag{6.14}$$

Equation (6.13) can be written recursively as:

$$f = f' \vee \pi_i)^{\alpha_i} \tag{6.15}$$

where

$$f' = \begin{cases} \pi_k^{\alpha_k} \\ (f' \vee \pi_i)^{\alpha_i} \end{cases} \tag{6.16}$$

Equation (6.15) can be written using the material implication operator as

$$f = \neg f' \rightarrow \pi_i)^{\alpha_i} \tag{6.17}$$

A factored form, is written as:

$$f = f_1 \rightarrow f_2 \tag{6.18}$$

where f_1 and f_2 are written in the form Equation (6.17) or Equation (6.18).

6.4.2 Material Implication Synthesis

The performance of the circuit is directed related to the number of material implications operations that must be performed. Thus, if a Boolean function is written as a sequence of material implication operations, it is desirable to have as few operations as needed. Different approaches aim to obtain Boolean expressions with minimized number of material implication operations (POIKONEN; LEHTONEN; LAIHO, 2012; TEODOROVIC; DAUTOVIC; MALBASA, 2013). An algorithm that resembles the Quine-McCluskey algorithm (MCCLUSKEY, 1956) is presented in (POIKONEN; LEHTONEN; LAIHO, 2012). In (TEODOROVIC; DAUTOVIC; MALBASA, 2013), better results are obtained using a graph-based approach.

Algorithm 11 presents an algorithm for memristor implication synthesis using FC, which uses material implication as the basic operation. It is considered that multiple input implication can be used (POIKONEN; LEHTONEN; LAIHO, 2012; TEODOROVIC; DAUTOVIC; MALBASA, 2013), which allows any π_k to be computed in one single step. The false constant and the π_k, where $1 \leq k < l$ have cost zero, which are created in the method CREATE_INITIAL_FUNCTIONS (line 4). It is not necessary to store π_l because $1 \rightarrow \pi = \pi$ and $\pi \rightarrow 1 = 1$.

FC-MEMRISTOR can generate both recursive or factored forms. For the recursive form, the cost is $C + 1$, where the cost is the number of implications, a function f_1 with cost C is combined with a function $f2$ with cost zero. This is performed by the method ASSOCIATE_REC (line 22-23).

For the factored form with cost $C + 1$, any two functions f_1 and f_2 can be combined as long the sum of their costs is C. Since the material implication is not a symmetric operation, it is necessary to combine the functions f_1 and f_2 in $f_1 \rightarrow f_2$ as well as $f_2 \rightarrow f_1$. This is performed by the ASSOCIATE_FACTOR method (line 14-20).

6.4.2.1 Memristor Counting

To utilize factored forms, it is necessary to determine the number of working memristor required. The approach proposed in this work consists of performing a tree traversal. A tree representing a factored form is directly obtained considering that each operation $p \rightarrow q$ generates a node $\rightarrow$ with left child p and right child q. Some additional nodes might be required to ensure that there no operation overwrites the value of an input memristor. That is, expressions of the type $\neg\pi_i \rightarrow \neg\pi_j$ are replaced by $\neg\pi_i \rightarrow ((\neg\pi_j \rightarrow 0) \rightarrow 0)$ if is not stored in a working memristor.

Algorithm 11 FC-MEMRISTOR Algorithm

```
1:  function CREATE_ALL_FUNCTIONS (n, factor)
2:      B ← ∅
3:      i ← 1
4:      B[0] ← CREATE_INITIAL_FUNCTIONS ()
5:      while any function is not synthesized do
6:          if factor then
7:              B[i] ← ASSOCIATE_FACTOR (B, i)
8:          else
9:              B[i] ← ASSOCIATE_REC (B, i)
10:         i ← i + 1
11:     return B
12:
13: function ASSOCIATE_FACTOR (B, i)
14:     S ← ∅
15:     for k ← 0, (i/2) do
16:         l ← i − k
17:         S ← S ∪ COMBINE (B[k], B[l], IMPL)
18:         S ← S ∪ COMBINE (B[l], B[k], IMPL)
19:     return S
20:
21: function ASSOCIATE_REC (B, i)
22:     S ← COMBINE (B[i − 1], B[0], IMPL)
23:     return S
```

That way all $\neg\pi_j$ nodes are leaves and are to the left of a $\rightarrow$ operator. All 0 nodes are leaves and are to the right of a $\rightarrow$ operator.

Algorithm 12 describes the factoring method and is divided into two submethods: MEMRISTOR-COUNT and REC. The MEMRISTOR-COUNT method initializes variables and starts the recursion, while rec traverses the tree. Two variables max and free are used (line 2 and 3). The variable max stores the maximum number of working memristors required and the variable free stores the number of working memristors that are free to be used to store the result of an operation. When a leaf node $\neg\pi_j$ is visited, the algorithm simply returns without modifying neither free nor max values (line 8). When a leaf node containing the constant zero is visited (line 9), the algorithm tests if there is a free memristor. If it is true, then the number of free memristors is reduced (line 10). Otherwise, there are no free memristors, and one work memristor must be added (line 11). When a material implication node is visited, the recursion is called for both children (line 14 and 15). After the recursions, it is checked if the memristor in the left can be reused. If this condition is true, the free variable is incremented and the recursion return true, since always one memristor can be reused (lines 16 and 17, respectively). To ensure the minimal number of memristor counting, different traversal orders must be considered.

Algorithm 12 Memristor Counting algorithm

```
1:  function MEMRISTOR-COUNT (noden)
2:      free ← 0
3:      max ← 0
4:      REC(n, &max, &free)
5:      return max
6:
7:  function REC (noden,* max,* free)
8:      if n is PI then return false
9:      else if n is 0 then
10:         if ((*free) > 0) then (*free) ← (*free) − 1
11:         else(*max) ←* max) + 1
12:             return true
13:     else
14:         canReuse = REC(n.left, max, free);
15:         REC(n.right, max, free);
16:         if canReuse is true then *free ←* free + 1
17:         return true
```

As example, the tree for the function described in Equation (6.19), which the factored form using material implication is presented in Equation (6.20), is shown in Figure 6.25. Performing a post-order traversal the first operation evaluated is $\neg\pi_2 \rightarrow 0$ which creates the first working memristor (M_1), incrementing max. Similarly, the second operation visited also creates a working memristor (M_2), incrementing both max and free since M1 can be used to store a different value. The third operation $\neg\pi_1 \rightarrow ((\neg\pi_2 \rightarrow 0) \rightarrow 0)$ directly uses M_2 to store the result. The next operation node visited computes $\neg\pi_4 \rightarrow 0$, which is located in the right side of the root node. For this node, M_1 is used to store the result, and free is decreased to zero. The next tree node creates a work memristor M3 because there are no free memristors. The remaining operations directly store the result on M_3.

$$f = (c \vee b) \wedge (\neg b \vee (\neg c \wedge \neg a)) \tag{6.19}$$

$$f = ((\neg\pi_1 \rightarrow \neg\pi_2) \rightarrow ((\neg\pi_1 \rightarrow (\neg\pi_2 \rightarrow \neg\pi_4)) \rightarrow 0)) \tag{6.20}$$

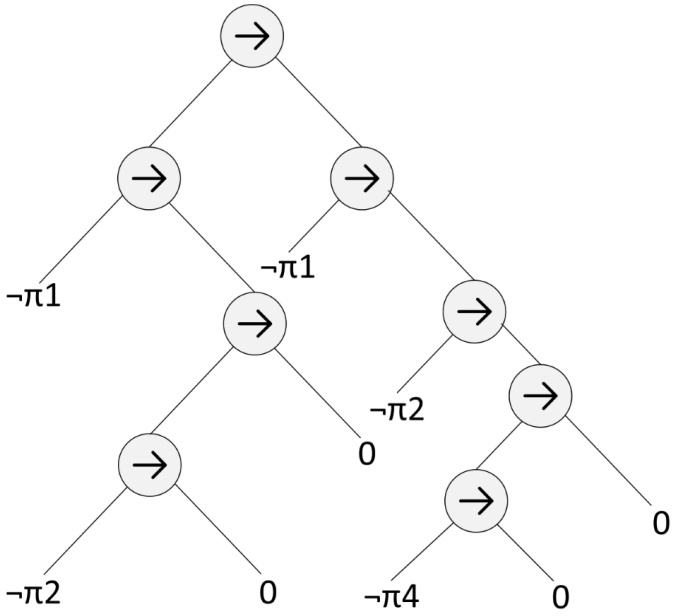


Figure 6.25 – Operator tree for Equation (6.20).

6.4.3 Results

All Boolean functions up to 4 inputs were synthesized using factored and recursive formulas. Table III compares the results to those presented in (1) (POIKONEN; LEHTONEN; LAIHO, 2012) and (2) (TEODOROVIC; DAUTOVIC; MALBASA, 2013) regarding the average number of implications operations. As expected, the utilization of factored forms reduced the number of implications operations.

Table 6.10 – Comparison of the average number of implications to implement all functions with at most 4 inputs (MARRANGHELLO et al., 2014a).

Methods	Avg. # of implications	Normalized average
This work (factored)	8.13	1.00
This work (recursive)	8.84	1.08
(1) (recursive)	8.91	1.10
(2) (recursive)	12.00	1.47

Table 6.11 presents the reduction in number of implications for all functions when the factored forms are used instead of recursive forms. The average number of operations is reduced in approximately 8%. Even though for some functions there is no gain using recursive forms, this gain is significant for other functions. For instance, the following expressions represent the same Boolean function:

Table 6.11 – Reduction in the number of implications operations of factored forms over recursive forms (MARRANGHELLO et al., 2014a).

Gain	# Functions	Gain	# Functions
0	31,537	4	204
1	22,729	5	51
2	9,614	6	12
3	1,389	**Total**	**65,536**

$$f = ((\neg\pi_{12} \rightarrow ((\neg\pi_9 \rightarrow ((\neg\pi_8 \rightarrow ((\neg\pi_2 \rightarrow ((\neg\pi_1 \rightarrow \neg\pi_4) \rightarrow 0)) \rightarrow 0)) \rightarrow 0)) \rightarrow 0) \rightarrow 0 \tag{6.21}$$

$$f = (\neg\pi_1 \rightarrow \neg\pi_4) \rightarrow (\neg\pi_2 \rightarrow \neg\pi_8) \rightarrow 0 \tag{6.22}$$

Equation (6.21) is a recursive form which requires 10 operations whereas the factored form is shown in Equation (6.22) requires only 4 operations. This represents the best case of 60% reduction in the number of required operations. It was observed that considering functions with at most 4 inputs, one extra memristor is required when using factored forms instead of recursive forms.

7 CONCLUSIONS

The main contribution of this thesis are 6 applications presented using functional composition (FC). FC is more straightforward to adapt to new technologies. It does require new algorithms and composition rules, but they are easier to compose, than to be targeted in a decomposition. All applications methods have been presented with promising results. The applications are: (*i*) multi-output factorization; (*ii*) synthesis of approximate functions for ATMR schemes; (*iii*) synthesis of threshold circuits; (*iv*) synthesis of majority circuits; (*v*) synthesis of spin-diode circuits; and (*vi*) synthesis using memristors.

The first application is a multi-output factorization algorithm using FC applied in an iterative remapping flow. This factorization algorithm can perform functional logic sharing and synthesize multiple outputs. For the benchmark circuits analyzed, results show that the flow can reduce combinational area up to 34% while still respecting the required timing. Also, the experiments have shown that the use of complex logic gates is not well explored by commercial tools since most of the optimizations performed by these tools are algebraic. The use of complex gates is possible to the Boolean optimizations carried out by the factorization algorithm, showing a higher quality of results with a larger amount of different combinational cells.

The second application proposed an approximation algorithm using FC to generate approximate functions and Full-ATMR modules. The approximate functions allow choosing a good balance between area overhead and fault coverage. The Full-ATMR allows a greater reduction in overhead costs and still can maintain a good protection ratio. The synthesis of approximate functions using FC allowed to find a balance between area and unprotected vectors (XOR between the approximate and original function). It was possible to keep the protected p-n junction ratio above 97% with only 125% area overhead. For the ATMR version of a 4-bit ripple-carry adder, several implementations are proposed, ranging from 93%/136% to 96%/168% of protected junctions and area overhead, respectively.

The third application explores FC on emerging technologies based on threshold logic, as RTD and STT-MTJ. The proposed method can consider multiple costs such as the number of threshold logic gate, logic depth, and number of interconnections. Experiments over MCNC benchmark circuits have shown that the threshold gate count, logic depth, and interconnections decreased 32%, 19.3% and 15.85% in average, respectively, compared to previous works.

The fourth application presented an algorithm was introduced for synthesizing circuits using majority and inverter gates, using FC, for technologies as QCA, SET, and TPL. Also, it presents the synthesis of AOI gates for the QCA technology. This algorithm generates an optimal

structure composed of majority or AOI gates for a given a Boolean function. The algorithm was applied to generate several libraries in an automated way: 4-input cell library; AOI library or an MAJ+AOI+INV library; then these libraries were used in mapping experiments. The results over MCNC benchmarks show that there is an average reduction up to 47% in area and 14% in logic depth, compared to previous works.

The fifth application introduced a set of algorithms for synthesizing circuits using spin-diodes using FC. The first algorithm generates an optimized spin-diode network considering logic sharing of all functions of four inputs. The second algorithm is a heuristic that can synthesize functions up to eight inputs. For functions with more than 9 inputs, a set of rules was developed to efficiently translate a factored form into a spin-diode tree. To evaluate the quality of these algorithms, they were used to transform a CMOS library into a spin-diode library. The results over OpenCores contained in IWLS 2005 benchmarks show that there is a combinational area reduction up to 50% in area and up to 30% in total area, compared to a minimal library composed of spin-diode basic gates.

The last application proposed the synthesis of factored forms together with the concept of multi-input implication in the logic synthesis for memristive IMPLY stateful logic using FC. As a result, the average number of IMPLY operations to perform a function with at most four inputs was reduced by 12% when compared to previous works, without requiring additional devices. Moreover, implementations of approximately 67% of the Boolean functions considered have been improved.

The proposed algorithms have been proved useful for optimization/synthesis, but there is a large space for optimization in the proposed algorithms yet. For instance, in the multi-output factorization method, an analysis to detect subfunctions can be implemented. For the generation of ATMR circuits, the technique can be expanded to approximate libraries, probably generating ATMR circuits with even less area. The threshold synthesis algorithm can be probably improved trying a logic sharing between threshold logic gates. Improvements in majority synthesis include considering MAJ5 gates, adopted in designs (NAVI et al., 2010). Future work for the spin-diode synthesis includes the reduction of sequential structures implemented using spin-diodes and expansion of the rules in the LST algorithm to be able to manipulate n-ary trees. Improvements in memristor synthesis include heuristic algorithms to synthesize functions with more inputs.

Also, other applications can exploit the FC principles and generate better results than known algorithms. An example is the double gate FinFET device, which is a promising device (CHANG et al., 2003; NOWAK et al., 2004). Preliminary synthesis' results show a reduction of 6% in number of devices, compared to (POSSANI et al., 2014).

REFERENCES

AKERS, S. B. Synthesis of combinational logic using three-input majority gates. In: IEEE. SWITCHING CIRCUIT THEORY AND LOGICAL DESIGN, 1962. SWCT 1962. PROCEEDINGS OF THE THIRD ANNUAL SYMPOSIUM ON. **Proceedings...** [S.l.], 1962. p. 149–158.

ALBRECHT, C. Iwls 2005 benchmarks. In: INTERNATIONAL WORKSHOP FOR LOGIC SYNTHESIS. **Proceedings...** [S.l.: s.n.], 2005.

ALPERT, C. J.; MEHTA, D. P.; SAPATNEKAR, S. S. **Handbook of algorithms for physical design automation**. [S.l.]: CRC press, 2008.

AMARU, L. et al. New logic synthesis as nanotechnology enabler. IEEE, 2015.

AMARU, L. et al. Multiple independent gate fets: How many gates do we need? In: IEEE. DESIGN AUTOMATION CONFERENCE (ASP-DAC), 2015 20TH ASIA AND SOUTH PACIFIC. **Proceedings...** [S.l.], 2015. p. 243–248.

ARO, V. C. et al. Read-polarity-once functions. **Logic & Synthesis, 21th International Workshop on**, Proceedings of IWLS, 2012.

ASHENHURST, R. L. The decomposition of switching functions. In: PROCEEDINGS OF AN INTERNATIONAL SYMPOSIUM ON THE THEORY OF SWITCHING, APRIL 1957. **Proceedings...** [S.l.: s.n.], 1957. p. 74–116.

AVEDILLO, M. J.; QUINTANA, J. M.; ROLDÁN, H. P. Increased logic functionality of clocked series-connected rtds. **Nanotechnology, IEEE Transactions on**, IEEE, v. 5, n. 5, p. 606–611, 2006.

AVERIN, D.; LIKHAREV, K. Coulomb blockade of single-electron tunneling, and coherent oscillations in small tunnel junctions. **Journal of low temperature physics**, Springer, v. 62, n. 3-4, p. 345–373, 1986.

BAUMANN, R. C. Radiation-induced soft errors in advanced semiconductor technologies. **Device and Materials Reliability, IEEE Transactions on**, IEEE, v. 5, n. 3, p. 305–316, 2005.

BEIU, V.; QUINTANA, J. M.; AVEDILLO, M. J. Vlsi implementations of threshold logic-a comprehensive survey. **Neural Networks, IEEE Transactions on**, IEEE, v. 14, n. 5, p. 1217–1243, 2003.

Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification, Release 20130425. In: . [S.l.: s.n.], 2013.

BERNSTEIN, G. H. et al. Magnetic qca systems. **Microelectronics Journal**, Elsevier, v. 36, n. 7, p. 619–624, 2005.

BERNSTEIN, K. et al. High-performance cmos variability in the 65-nm regime and beyond. **IBM journal of research and development**, IBM, v. 50, n. 4.5, p. 433–449, 2006.

BERTACCO, V.; DAMIANI, M. The disjunctive decomposition of logic functions. In: IEEE COMPUTER SOCIETY. PROCEEDINGS OF THE 1997 IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings...** [S.l.], 1997. p. 78–82.

BORGHETTI, J. et al. Memristive switches enable ‘stateful’logic operations via material implication. **Nature**, Nature Publishing Group, v. 464, n. 7290, p. 873–876, 2010.

BORKAR, S. Design perspectives on 22nm cmos and beyond. In: ACM. PROCEEDINGS OF THE 46TH ANNUAL DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 2009. p. 93–94.

BRAYTON, R.; MISHCHENKO, A. Abc: An academic industrial-strength verification tool. In: SPRINGER. COMPUTER AIDED VERIFICATION. **Proceedings...** [S.l.], 2010. p. 24–40.

BRAYTON, R. K. Factoring logic functions. **IBM Journal of Research and Development**, IBM, v. 31, n. 2, p. 187–198, 1987.

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

BRAYTON, R. K. et al. Multi-level logic optimization and the rectangular covering problem. In: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON COMPUTE AIDED DESIGN. **Proceedings...** [S.l.: s.n.], 1987.

CALHOUN, B. B. H. et al. Digital circuit design challenges and opportunities in the era of nanoscale cmos. **Proceedings of the IEEE**, IEEE, v. 96, n. 2, p. 343–365, 2008.

CALLEGARO MAYLER GA MARTINS, R. P. R. A. I. R. V. Dsd synthesis based on variable intersection graphs. In: 30TH SOUTH SYMPOSIUM ON MICROELECTRONICS. **Proceedings...** [S.l.: s.n.], 2015.

CALLEGARO, V. et al. Bottom-up disjoint-support decomposition based on cofactor and boolean difference analysis. In: IEEE. COMPUTER DESIGN (ICCD), 2015 IEEE INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.], 2015.

CALLEGARO, V. et al. Read-polarity-once boolean functions. In: IEEE. INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI), 2013 26TH SYMPOSIUM ON. **Proceedings...** [S.l.], 2013. p. 1–6.

CALLEGARO, V. et al. Read-polarity-once boolean functions revisited. In: PROCEEDINGS OF IWLS. LOGIC & SYNTHESIS, 23TH INTERNATIONAL WORKSHOP ON. **Proceedings...** [S.l.], 2013.

CALLEGARO, V. et al. A domain-transformation approach to synthesize read-polarity-once boolean functions. **Journal of Integrated Circuits and Systems**, v. 9, n. 1, p. 60–69, 2014.

CHANG, L. et al. Extremely scaled silicon nano-cmos devices. **Proceedings of the IEEE**, IEEE, v. 91, n. 11, p. 1860–1873, 2003.

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

CHOI, S. et al. A novel high-speed multiplexing ic based on resonant tunneling diodes. **Nanotechnology, IEEE Transactions on**, IEEE, v. 8, n. 4, p. 482–486, 2009.

CHOI, Y.-K. et al. Sub-20nm cmos finfet technologies. IEEE, 2001.

CHOUDHURY, M.; MOHANRAM, K. Bi-decomposition of large boolean functions using blocking edge graphs. In: IEEE PRESS. PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings...** [S.l.], 2010. p. 586–591.

CHUA, L. O. Memristor-the missing circuit element. **Circuit Theory, IEEE Transactions on**, IEEE, v. 18, n. 5, p. 507–519, 1971.

CHUA, L. O.; KANG, S. M. Memristive devices and systems. **Proceedings of the IEEE**, IEEE, v. 64, n. 2, p. 209–223, 1976.

CONG, J.; DING, Y. Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, IEEE, v. 13, n. 1, p. 1–12, 1994.

CORREIA, V.; REIS, A. Advanced technology mapping for standard-cell generators. In: ACM. PROCEEDINGS OF THE 17TH SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN. **Proceedings...** [S.l.], 2004. p. 254–259.

COUDERT, O. Two-level logic minimization: an overview. **Integration, the VLSI journal**, Elsevier, v. 17, n. 2, p. 97–140, 1994.

CURTIS, H. A. **A new approach to the design of switching circuits**. [S.l.]: van Nostrand, 1962.

DENNARD, R. H. Past progress and future challenges in lsi technology: From dram and scaling to ultra-low-power cmos. **Solid-State Circuits Magazine, IEEE**, IEEE, v. 7, n. 2, p. 29–38, 2015.

DETJENS, E. et al. Technology mapping in mis. In: PROC. OF THE ICCAD. **Proceedings...** [S.l.: s.n.], 1987. v. 87, p. 116–119.

ENTRENA, L. et al. Logic masking for set mitigation using approximate logic circuits. In: IEEE. ON-LINE TESTING SYMPOSIUM (IOLTS), 2012 IEEE 18TH INTERNATIONAL. **Proceedings...** [S.l.], 2012. p. 176–181.

FAHMY, H.; KIEHL, R. A. Complete logic family using tunnelingphase-logic devices. In: CITESEER. PROC. INT. CONF. MICROELECTRON. **Proceedings...** [S.l.], 1999. p. 22–24.

FRIEDMAN, J. S. et al. Inmnas magnetoresistive spin-diode logic. In: ACM. PROCEEDINGS OF THE GREAT LAKES SYMPOSIUM ON VLSI. **Proceedings...** [S.l.], 2012. p. 209–214.

FRIEDMAN, J. S. et al. A spin-diode logic family. **IEEE Trans. Nanotechnol.**, IEEE, v. 11, n. 5, p. 1026–1032, 2012.

GANG, Y. et al. A high-reliability, low-power magnetic full adder. **Magnetics, IEEE Transactions on**, IEEE, v. 47, n. 11, p. 4611–4616, 2011.

GIELEN, G. et al. Emerging yield and reliability challenges in nanometer cmos technologies. In: ACM. PROCEEDINGS OF THE CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE. **Proceedings...** [S.l.], 2008. p. 1322–1327.

GOMES, I.; KASTENSMIDT, F. G. et al. Reducing tmr overhead by combining approximate circuit, transistor topology and input permutation approaches. In: IEEE. INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI), 2013 26TH SYMPOSIUM ON. **Proceedings...** [S.l.], 2013. p. 1–6.

GOMES, I. et al. Methodology for achieving best trade-off of area and fault masking coverage in atmr. In: IEEE. TEST WORKSHOP-LATW, 2014 15TH LATIN AMERICAN. **Proceedings...** [S.l.], 2014. p. 1–6.

GOMES, I. et al. Using only redundant modules with approximate logic to reduce drastically area overhead in tmr. In: IEEE. TEST SYMPOSIUM (LATS), 2015 16TH LATIN-AMERICAN. **Proceedings...** [S.l.], 2015. p. 1–6.

GOMES, I. A. et al. Exploring the use of approximate tmr to mask transient faults in logic with low area overhead. **Microelectronics Reliability**, Pergamon, 2015.

GOWDA, T. et al. Identification of threshold functions and synthesis of threshold networks. **Computer-aided design of integrated circuits and systems, IEEE transactions on**, IEEE, v. 30, n. 5, p. 665–677, 2011.

HLAVIČKA, J.; FIŠER, P. Boom: a heuristic boolean minimizer. In: IEEE PRESS. PROCEEDINGS OF THE 2001 IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings...** [S.l.], 2001. p. 439–442.

ITRS. **Int'l Tech. Roadmap for Semiconductors**. 2015. Http://www.itrs2.net/itrs-reports.html.

JÓŹWIAK, L. Information relationships and measures in application to logic design. In: IEEE. MULTIPLE-VALUED LOGIC, 1999. PROCEEDINGS. 1999 29TH IEEE INTERNATIONAL SYMPOSIUM ON. **Proceedings...** [S.l.], 1999. p. 228–235.

JÓZWIAK, L.; BIEGAŃSKI, S. Technology library modelling for information-driven circuit synthesis. In: IEEE. DIGITAL SYSTEM DESIGN ARCHITECTURES, METHODS AND TOOLS, 2008. DSD'08. 11TH EUROMICRO CONFERENCE ON. **Proceedings...** [S.l.], 2008. p. 480–489.

KEUTZER, K. Dagon: technology binding and local optimization by dag matching. In: ACM. PAPERS ON TWENTY-FIVE YEARS OF ELECTRONIC DESIGN AUTOMATION. **Proceedings...** [S.l.], 1988. p. 617–624.

KILBY, J. S. **Semiconductor structure fabrication**. [S.l.]: Google Patents, 1963. US Patent 3,072,832.

KIM, K.; SHIN, S.; KANG, S.-M. Field programmable stateful logic array. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, IEEE, v. 30, n. 12, p. 1800–1813, 2011.

KONG, K.; SHANG, Y.; LU, R. An optimized majority logic synthesis methodology for quantum-dot cellular automata. **Nanotechnology, IEEE Transactions on**, IEEE, v. 9, n. 2, p. 170–183, 2010.

KOOMEY, J. G. et al. Implications of historical trends in the electrical efficiency of computing. **Annals of the History of Computing, IEEE**, IEEE, v. 33, n. 3, p. 46–54, 2011.

KUHN, K. J. Considerations for ultimate cmos scaling. **IEEE Trans. Electron Devices**, v. 59, n. 7, p. 1813–1828, 2012.

KUKIMOTO, Y.; BRAYTON, R. K.; SAWKAR, P. Delay-optimal technology mapping by dag covering. In: ACM. PROCEEDINGS OF THE 35TH ANNUAL DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 1998. p. 348–351.

KVATINSKY, S. et al. Memristor-based material implication (imply) logic: Design principles and methodologies. **Very Large Scale Integration (VLSI) Systems, IEEE Transactions on**, IEEE, v. 22, n. 10, p. 2054–2066, 2014.

LAIHO, M.; LEHTONEN, E. Cellular nanoscale network cell with memristors for local implication logic and synapses. In: IEEE. CIRCUITS AND SYSTEMS (ISCAS), PROCEEDINGS OF 2010 IEEE INTERNATIONAL SYMPOSIUM ON. **Proceedings...** [S.l.], 2010. p. 2051–2054.

LEE, T.-L.; WANG, C.-Y. Recognition of fanout-free functions. In: IEEE COMPUTER SOCIETY. PROCEEDINGS OF THE 2007 ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 2007. p. 426–431.

LEHTONEN, E.; LAIHO, M. Stateful implication logic with memristors. In: IEEE COMPUTER SOCIETY. PROCEEDINGS OF THE 2009 IEEE/ACM INTERNATIONAL SYMPOSIUM ON NANOSCALE ARCHITECTURES. **Proceedings...** [S.l.], 2009. p. 33–36.

LEHTONEN, E.; POIKONEN, J. H.; LAIHO, M. Two memristors suffice to compute all boolean functions. **Electronics letters**, v. 46, n. 3, p. 230, 2010.

LENT, C. S.; ISAKSEN, B.; LIEBERMAN, M. Molecular quantum-dot cellular automata. **Journal of the American Chemical Society**, ACS Publications, v. 125, n. 4, p. 1056–1063, 2003.

LENT, C. S. et al. Quantum cellular automata. **Nanotechnology**, IOP Publishing, v. 4, n. 1, p. 49, 1993.

LERCH, J. **Threshold gate circuits employing field-effect transistors**. [S.l.]: Google Patents, 1973. US Patent 3,715,603.

LITVINOV, V. et al. Resonant tunneling in iii-nitrides. **Proceedings of the IEEE**, IEEE, v. 98, n. 7, p. 1249–1254, 2010.

MACHADO, L. et al. Kl-cut based digital circuit remapping. In: IEEE. NORCHIP, 2012. **Proceedings...** [S.l.], 2012. p. 1–4.

MACHADO, L. et al. Iterative remapping respecting timing constraints. In: BEST OF ISVLSI 2013. **Proceedings...** [S.l.: s.n.], forthcoming.

MACHADO, L. et al. Iterative remapping respecting timing constraints. In: IEEE. VLSI (ISVLSI), 2013 IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON. **Proceedings...** [S.l.], 2013. p. 236–241.

MAHMOUDI, H. et al. Implication logic gates using spin-transfer-torque-operated magnetic tunnel junctions for intrinsic logic-in-memory. **Solid-State Electronics**, Elsevier, v. 84, p. 191–197, 2013.

MAHMOUDI, H. et al. Reliability analysis and comparison of implication and reprogrammable logic gates in magnetic tunnel junction logic circuits. **Magnetics, IEEE Transactions on**, IEEE, v. 49, n. 12, p. 5620–5628, 2013.

MAILHOT, F.; MICHELI, G. D. Technology mapping using boolean matching and don't care sets. In: IEEE COMPUTER SOCIETY PRESS. PROCEEDINGS OF THE CONFERENCE ON EUROPEAN DESIGN AUTOMATION. **Proceedings...** [S.l.], 1990. p. 212–216.

MARRANGHELLO, F. S. et al. Factored forms for memristive material implication stateful logic. **IEEE Journal on Emerging and Selected Topics in Circuits and Systems**, IEEE, v. 5, n. 2, p. 267–278, 2015.

MARRANGHELLO, F. S. et al. Improved logic synthesis for memristive stateful logic using multi-memristor implication. In: IEEE. CIRCUITS AND SYSTEMS (ISCAS), 2015 IEEE INTERNATIONAL SYMPOSIUM ON. **Proceedings...** [S.l.], 2015. p. 181–184.

MARRANGHELLO, F. S. et al. Exploring factored forms for sequential implication logic synthesis. In: IEEE. NANOTECHNOLOGY (IEEE-NANO), 2014 IEEE 14TH INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.], 2014. p. 268–273.

MARRANGHELLO, F. S. et al. Comparison of recursive and factored forms for memristor based implication logic. In: PROCEEDINGS OF IWLS. LOGIC & SYNTHESIS, 23TH INTERNATIONAL WORKSHOP ON. **Proceedings...** [S.l.], 2014.

MARTINELLO, O. et al. Kl-cuts: a new approach for logic synthesis targeting multiple output blocks. In: EUROPEAN DESIGN AND AUTOMATION ASSOCIATION. PROCEEDINGS OF CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE. **Proceedings...** [S.l.], 2010. p. 777–782.

MARTINS, M. et al. Open cell library in 15nm freepdk technology. **Proc. of the Int'l Symp. on Physical Design (ISPD)**, 2015.

MARTINS, M. G. et al. Majority-based library generation for qca, set and tpl technologies. In: 28TH SOUTH SYMPOSIUM ON MICROELECTRONICS. **Proceedings...** [S.l.: s.n.], 2013.

MARTINS, M. G. et al. Majority-based logic synthesis for nanometric technologies. In: IEEE. NANOTECHNOLOGY (IEEE-NANO), 2014 IEEE 14TH INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.], 2014. p. 256–261.

MARTINS, M. G. et al. Computing minimum decision chains of boolean functions. In: 26TH SOUTH SYMPOSIUM ON MICROELECTRONICS. **Proceedings...** [S.l.: s.n.], 2011.

MARTINS, M. G. et al. Spin diode network synthesis using functional composition. In: IEEE. INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI), 2013 26TH SYMPOSIUM ON. **Proceedings...** [S.l.], 2013. p. 1–6.

MARTINS, M. G.; RIBAS, R. P.; REIS, A. I. Applications of functional composition. In: 27TH SOUTH SYMPOSIUM ON MICROELECTRONICS. **Proceedings...** [S.l.: s.n.], 2012.

MARTINS, M. G. A. et al. Functional composition paradigm and applications. In: PROCEEDINGS OF IWLS. LOGIC & SYNTHESIS, 22TH INTERNATIONAL WORKSHOP ON. **Proceedings...** [S.l.], 2012.

MARTINS, M. G. A. et al. Majority logic synthesis for nanometric technologies. In: PROCEEDINGS OF IWLS. LOGIC & SYNTHESIS, 23TH INTERNATIONAL WORKSHOP ON. **Proceedings...** [S.l.], 2014.

MARTINS, M. G. A. et al. Efficient method to compute minimum decision chains of boolean functions. In: ACM. PROCEEDINGS OF THE 21ST EDITION OF THE GREAT LAKES SYMPOSIUM ON GREAT LAKES SYMPOSIUM ON VLSI. **Proceedings...** [S.l.], 2011. p. 419–422.

MARTINS, M. G. A. et al. Boolean factoring with multi-objective goals. In: IEEE. COMPUTER DESIGN (ICCD), 2010 IEEE INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.], 2010. p. 229–234.

MARTINS, M. G. A. et al. Automated synthesis approaches for digital integrated design of spin-diode circuits. In: PROCEEDINGS OF IWLS. LOGIC & SYNTHESIS, 24TH INTERNATIONAL WORKSHOP ON. **Proceedings...** [S.l.], 2015.

MARTINS, M. G. A. et al. Enhanced spin-diode synthesis using logic sharing. In: DIGITAL SYSTEM DESIGN (DSD), 2015 EUROMICRO CONFERENCE ON. DIGITAL SYSTEM DESIGN (DSD), 2015 18TH EUROMICRO CONFERENCE ON. **Proceedings...** [S.l.], 2015. p. 218–224.

MARTINS, M. G. A.; RIBAS, R. P.; REIS, A. Functional composition: A new paradigm for performing logic synthesis. In: IEEE. QUALITY ELECTRONIC DESIGN (ISQED), 2012 13TH INTERNATIONAL SYMPOSIUM ON. **Proceedings...** [S.l.], 2012. p. 236–242.

MAY, S.; WESSELS, B. High-field magnetoresistance in p-(in, mn) as/n-inas heterojunctions. **Applied physics letters**, AIP, v. 88, n. 7, p. 072105–072105, 2006.

MCCLUSKEY, E. J. Minimization of boolean functions*. **Bell system technical Journal**, Wiley Online Library, v. 35, n. 6, p. 1417–1444, 1956.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. **The bulletin of mathematical biophysics**, Springer, v. 5, n. 4, p. 115–133, 1943.

MICHELI, G. D. **Synthesis and optimization of digital circuits**. [S.l.]: McGraw-Hill Higher Education, 1994.

MILLER, H.; WINDER, R. Majority-logic synthesis by geometric methods. **IEEE Transactions on Electronic Computers**, v. 1, n. EC-11, p. 89–90, 1962.

MINATO, S.-i.; MICHELI, G. D. Finding all simple disjunctive decompositions using irredundant sum-of-products forms. In: ACM. PROCEEDINGS OF THE 1998 IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings...** [S.l.], 1998. p. 111–117.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. Dag-aware aig rewriting a fresh look at combinational logic synthesis. In: ACM. PROCEEDINGS OF THE 43RD ANNUAL DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 2006. p. 532–535.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. K. Improvements to technology mapping for lut-based fpgas. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, IEEE, v. 26, n. 2, p. 240–253, 2007.

MISHCHENKO, A. et al. **FRAIGs: A unifying representation for logic synthesis and verification**. [S.l.], 2005.

MISHCHENKO, A.; STEINBACH, B.; PERKOWSKI, M. An algorithm for bi-decomposition of logic functions. In: ACM. PROCEEDINGS OF THE 38TH ANNUAL DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 2001. p. 103–108.

MOMENZADEH, M. et al. Characterization, test, and logic synthesis of and-or-inverter (aoi) gate design for qca implementation. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, IEEE, v. 24, n. 12, p. 1881–1893, 2005.

MOORE, G. E. Cramming more components onto integrated circuits. **Electronics**, IEEE, v. 38, n. 8, p. 114–117, 1965.

MOREIRA, M. et al. Semi-custom ncl design with commercial eda frameworks: Is it possible? In: IEEE. ASYNCHRONOUS CIRCUITS AND SYSTEMS (ASYNC), 2014 20TH IEEE INTERNATIONAL SYMPOSIUM ON. **Proceedings...** [S.l.], 2014. p. 53–60.

MUROGA, S. Threshold logic and its applications. 1971.

NAVI, K. et al. Five-input majority gate, a new device for quantum-dot cellular automata. **Journal of Computational and Theoretical Nanoscience**, American Scientific Publishers, v. 7, n. 8, p. 1546–1553, 2010.

NEUMANN, J. V. Probabilistic logics and the synthesis of reliable organisms from unreliable components. **Automata studies**, v. 34, p. 43–98, 1956.

NEUTZLING, A. et al. An efficient method to threshold logic functions identification. In: 28TH SOUTH SYMPOSIUM ON MICROELECTRONICS. **Proceedings...** [S.l.: s.n.], 2013.

NEUTZLING, A. et al. Synthesis of threshold logic gates to nanoelectronics. In: IEEE. INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI), 2013 26TH SYMPOSIUM ON. **Proceedings...** [S.l.], 2013. p. 1–6.

NEUTZLING, A. et al. A constructive approach for threshold logic circuit synthesis. In: IEEE. CIRCUITS AND SYSTEMS (ISCAS), 2014 IEEE INTERNATIONAL SYMPOSIUM ON. **Proceedings...** [S.l.], 2014. p. 385–388.

NEUTZLING, A. et al. Threshold logic synthesis using functional composition paradigm. **Logic & Synthesis, 22th International Workshop on**, Proceedings of IWLS, 2013.

NEUTZLING, A. et al. An isop-based method for threshold logic identification. In: PROCEEDINGS OF IWLS. LOGIC & SYNTHESIS, 23TH INTERNATIONAL WORKSHOP ON. **Proceedings...** [S.l.], 2014.

NEUTZLING, A. et al. Factored forms for memristive material implication stateful logic. **ACM Journal on Emerging Technologies in Computing Systems**, submitted.

NOWAK, E. J. et al. Turning silicon on its edge [double gate cmos/finfet technology]. **Circuits and Devices Magazine, IEEE**, IEEE, v. 20, n. 1, p. 20–31, 2004.

NUKALA, N. S.; KULKARNI, N.; VRUDHULA, S. Spintronic threshold logic array (stla)—a compact, low leakage, non-volatile gate array architecture. **Journal of Parallel and Distributed Computing**, Elsevier, v. 74, n. 6, p. 2452–2460, 2014.

PACHA, C. et al. Circuit design issues in multi-gate fet cmos technologies. In: IEEE. SOLID-STATE CIRCUITS CONFERENCE, 2006. ISSCC 2006. DIGEST OF TECHNICAL PAPERS. IEEE INTERNATIONAL. **Proceedings...** [S.l.], 2006. p. 1656–1665.

PATIL, S. et al. Spintronic logic gates for spintronic data using magnetic tunnel junctions. In: IEEE. COMPUTER DESIGN (ICCD), 2010 IEEE INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.], 2010. p. 125–131.

PERSHIN, Y. V.; VENTRA, M. D. Neuromorphic, digital, and quantum computation with memory circuit elements. **Proceedings of the IEEE**, IEEE, v. 100, n. 6, p. 2071–2080, 2012.

PETERS, J. et al. Spin-dependent magnetotransport in a p-inmnsb/n-insb magnetic semiconductor heterojunction. **Applied Physics Letters**, AIP Publishing, v. 98, n. 19, p. 193506, 2011.

PETTENGHI, H.; AVEDILLO, M. J.; QUINTANA, J. M. Using multi-threshold threshold gates in rtd-based logic design: A case study. **Microelectronics Journal**, Elsevier, v. 39, n. 2, p. 241–247, 2008.

POIKONEN, J. H.; LEHTONEN, E.; LAIHO, M. On synthesis of boolean expressions for memristive devices using sequential implication logic. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, IEEE, v. 31, n. 7, p. 1129–1134, 2012.

POSSANI, V. N. et al. Exploring independent gates in finfet-based transistor network generation. In: ACM. PROCEEDINGS OF THE 27TH SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN. **Proceedings...** [S.l.], 2014. p. 41.

PRODROMAKIS, T.; TOUMAZOU, C.; CHUA, L. Two centuries of memristors. **Nature materials**, Nature Publishing Group, v. 11, n. 6, p. 478–481, 2012.

PUGGELLI, A. et al. Are logic synthesis tools robust? In: IEEE. DESIGN AUTOMATION CONFERENCE (DAC), 2011 48TH ACM/EDAC/IEEE. **Proceedings...** [S.l.], 2011. p. 633–638.

RAJENDRAN, J. et al. An energy-efficient memristive threshold logic circuit. **Computers, IEEE Transactions on**, IEEE, v. 61, n. 4, p. 474–487, 2012.

RANGARAJU, N.; LI, P.; WESSELS, B. Giant magnetoresistance of magnetic semiconductor heterojunctions. **Physical Review B**, APS, v. 79, n. 20, p. 205209, 2009.

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

SASAO, T. Fpga design by generalized functional decomposition. In: LOGIC SYNTHESIS AND OPTIMIZATION. **Proceedings...** [S.l.]: Springer, 1993. p. 233–258.

SASAO, T.; MATSUURA, M. Decompos: An integrated system for functional decomposition. In: 1998 INTERNATIONAL WORKSHOP ON LOGIC SYNTHESIS. **Proceedings...** [S.l.: s.n.], 1998. p. 471–477.

SCHNEIDER, F. R. et al. Exact lower bound for the number of switches in series to implement a combinational logic cell. In: IEEE. COMPUTER DESIGN: VLSI IN COMPUTERS AND PROCESSORS, 2005. ICCD 2005. PROCEEDINGS. 2005 IEEE INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.], 2005. p. 357–362.

SENTOVICH, E. M. et al. Sis: A system for sequential circuit synthesis. Citeseer, 1992.

SHANNON, C. et al. The synthesis of two-terminal switching circuits. **Bell System Technical Journal**, Wiley Online Library, v. 28, n. 1, p. 59–98, 1949.

SIERAWSKI, B. D.; BHUVA, B. L.; MASSENGILL, L. W. Reducing soft error rate in logic circuits through approximate logic functions. **Nuclear Science, IEEE Transactions on**, IEEE, v. 53, n. 6, p. 3417–3421, 2006.

SILVA, A. N. **Synthesis of Threshold Logic Based Circuits**. Dissertation (Master) — Universidade Federal do Rio Grande do Sul, 2014.

STRUKOV, D. B. et al. The missing memristor found. **nature**, Nature Publishing Group, v. 453, n. 7191, p. 80–83, 2008.

SUBIRATS, J. L.; JEREZ, J. M.; FRANCO, L. A new decomposition algorithm for threshold synthesis and generalization of boolean functions. **Circuits and systems I: Regular papers, IEEE Transactions on**, IEEE, v. 55, n. 10, p. 3188–3196, 2008.

TAUB, H. Digital integrated electronics. 1977.

TEODOROVIC, P.; DAUTOVIC, S.; MALBASA, V. Recursive boolean formula minimization algorithms for implication logic. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, IEEE, v. 32, n. 11, p. 1829–1833, 2013.

TSUI, C.-y. Technology dependent optimization for low power. In: LOGIC SYNTHESIS FOR LOW POWER VLSI DESIGNS. **Proceedings...** [S.l.]: Springer, 1998. p. 149–182.

VENKATARAMANI, S. et al. Salsa: systematic logic synthesis of approximate circuits. In: ACM. PROCEEDINGS OF THE 49TH ANNUAL DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 2012. p. 796–801.

WAGNER, F.; REIS, A.; RIBAS, R. **Fundamentos de circuitos digitais**. Porto Alegre: Sagra Luzzatto, 2006.

WALLMARK, J. T.; MARCUS, S. M. Minimum size and maximum packing density of nonredundant semiconductor devices. **Proceedings of the IRE**, IEEE, v. 50, n. 3, p. 286–298, 1962.

WANG, P. et al. Comprehensive majority/minority logic synthesis method. In: IEEE. NANOTECHNOLOGY (IEEE-NANO), 2013 13TH IEEE CONFERENCE ON. **Proceedings...** [S.l.], 2013. p. 694–697.

WULF, W. A.; MCKEE, S. A. Hitting the memory wall: implications of the obvious. **ACM SIGARCH computer architecture news**, ACM, v. 23, n. 1, p. 20–24, 1995.

YANG, C.; CIESIELSKI, M. Bds: a bdd-based logic optimization system. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, IEEE, v. 21, n. 7, p. 866–876, 2002.

YANG, S. **Logic synthesis and optimization benchmarks user guide: version 3.0**. [S.l.]: Microelectronics Center of North Carolina (MCNC), 1991.

YONEDA, I. et al. Study of nanoimprint lithography for applications toward 22nm node cmos devices. In: INTERNATIONAL SOCIETY FOR OPTICS AND PHOTONICS. SPIE ADVANCED LITHOGRAPHY. **Proceedings...** [S.l.], 2008. p. 692104–692104.

ZHANG, R.; GUPTA, P.; JHA, N. K. Synthesis of majority and minority networks and its applications to qca, tpl and set based nanotechnologies. In: IEEE. VLSI DESIGN, 2005. 18TH INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.], 2005. p. 229–234.

ZHANG, R. et al. Threshold network synthesis and optimization and its application to nanotechnologies. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, IEEE, v. 24, n. 1, p. 107–118, 2005.

ZHANG, R. et al. A method of majority logic reduction for quantum cellular automata. **Nanotechnology, IEEE Transactions on**, IEEE, v. 3, n. 4, p. 443–450, 2004.

ZHAO, W.; BELHAIRE, E.; CHAPPERT, C. Spin-mtj based non-volatile flip-flop. In: IEEE. NANOTECHNOLOGY, 2007. IEEE-NANO 2007. 7TH IEEE CONFERENCE ON. **Proceedings...** [S.l.], 2007. p. 399–402.

ZHAO, W.; CAO, Y. New generation of predictive technology model for sub-45 nm early design exploration. **Electron Devices, IEEE Transactions on**, IEEE, v. 53, n. 11, p. 2816–2823, 2006.

ZHU, X. et al. Performing stateful logic on memristor memory. **Circuits and Systems II: Express Briefs, IEEE Transactions on**, IEEE, v. 60, n. 10, p. 682–686, 2013.