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

MAYLER GAMA ALVARENGA MARTINS

Functional Composition and Applications

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

Prof. Dr. André Inácio Reis
Advisor

Prof. Dr. Renato Perez Ribas
Co-advisor

Porto Alegre, August 2012

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Gama Alvarenga Martins, Mayler

Functional Composition and Applications / Mayler Gama Alvarenga Martins. – Porto Alegre: PGMICRO da UFRGS, 2012.

97f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2012. Advisor: André Inácio Reis; Coadvisor: Renato Perez Ribas.

1. Boolean function. 2. Logic synthesis. 3. Functional composition. 4. Minimum decision chain. 5. Boolean factoring. 6. Exclusive-OR. 7. Majority gate. I. Reis, André Inácio. II. Ribas, Renato Perez. III. Functional Composition and Applications.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Pró-Reitor de Coordenação Acadêmica: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitor Adjunto de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Flavio Rech Wagner
Coordenador do PGMICRO: Prof. Ricardo Augusto da Luz Reis
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

ACKNOWLEDGMENTS

There are some people I wish to say “thank you”.

I would like to thank my advisor, André I. Reis, for his vast knowledge in logic synthesis and by often pointing out the path to be followed, even with my uncertainty about where this path would lead. Now I see that these paths led in the right direction.

I would like to thank my co-advisor, Renato P. Ribas, for its ability to organize ideas and texts, and his enthusiasm to talk about any subject. Thanks for all the barbecues held at his home.

I would like to thank the rest of my thesis committee: Prof. Sérgio Bampi, Prof. Fabrizio Ferrandi, and Prof. Francesc Echeto for their insightful comments.

To all labmates, by the intense discussions, that helped me in the development of this thesis. I want to thank (in order of reviews): Felipe Marranghelo, Lucas Machado, Vinicius Dal Bem and Vinicius Callegaro for all thesis’ reviews. These reviews improved a lot this text. I would like to thank in particular Vinicius Callegaro for your patience in helping me with the code issues, and by random discussions, ranging from brewing to poker strategies.

I owe my gratitude to my parents, Luiz C. P. Martins and Maruza G. Alvarenga, and my godmother Ediméa C. Gama, through the encouragement and support. I am grateful to my girlfriend Francine B. Puchalski for all the love and unconditional patience.

And finally, I would like to thank all Nangate A/S for financing my trip to Copenhagen in 2010 and given me the opportunity to show my work. I want to thank also CAPES funding agency and the European Community’s Seventh Framework Programme under grant 248538 – Synaptic. Without their investment, this work would not be possible.

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

CONTENTS

LIST OF ACRONYMS AND ABBREVIATIONS

AIG	And-Inverter Graph
BDD	Binary Decision Diagram
CMOS	Complementary Metal Oxide Semiconductor
DFT	Design for Testability
DSD	Disjoint Support Decomposition
EDA	Electronic Design Automation
ESOP	Exclusive-OR Sum Of Products
FC	Functional Composition
FD	Functional Decomposition
FPRM	Fixed Polarity Reed-Muller Expression
GF	Good Factor
GRM	Generalized Reed-Muller Expression
IC	Integrated Circuit
ISOP	Irredundant Sum-Of-Products
KRO	Kronecker Expression
MDC	Minimum Decision Chain
MF	Majority Function
NSP	Non Series-Parallel
OBDD	Ordered Binary Decision Diagram
PKDD	Pseudo-Kronecker Decision Diagram
PLA	Programmable Logic Array
POS	Product-Of-Sums
PPRM	Positive Polarity Reed-Muller Expression
PSDKRO	Pseudo Kronecker Expression
PSDRM	Pseudo Reed-Muller Expression
QBF	Quantified Boolean Formula
QF	Quick Factor

ROBDD	Reduced and Ordered Binary Decision Diagram
RTL	Register-Transfer Level
SET	Single Electron Tunneling
SPFD	Sets of Pairs of Functions to be Distinguished
SOP	Sum-Of-Products
TPL	Tunneling Phase Logic
VHDL	VHSIC Hardware Description Language
XF	X-Factor

LIST OF FIGURES

LIST DE TABLES

ABSTRACT

This work presents functional composition (FC) as a new paradigm for combinational logic synthesis. FC is a bottom-up approach to synthesize Boolean functions, being able to evaluate the cost of intermediate sub-functions, exploring a larger number of different candidate combinations. These are interesting advantages when compared to the top-down behavior of functional decomposition. FC presents great flexibility to implement algorithms with optimal or suboptimal results for different applications. The proposed strategy presents good results for the synthesis of Boolean functions targeting different technologies. FC is based on the following principles: (1) the representation of logic functions is done by a bonded pair of functional and structural representations; (2) the algorithm starts from a set of initial functions; (3) simpler functions are associated to create more complex ones; (4) there is a partial order, enabling dynamic programming; (5) a set of allowed functions can be used in order to reduce execution time/memory consumption. This work presents functional composition algorithms for Boolean factoring, including optimal factoring, Boolean factoring considering the exclusive-OR operator, minimum decision chain computation and synthesis of functions considering only majority and inverter logic gates.

Keywords: Boolean functions, logic synthesis, functional composition, minimum decision chain, Boolean factoring, exclusive-OR, majority gate.

Composição Funcional e Aplicações

RESUMO

Este trabalho apresenta a composição funcional (CF) como um novo paradigma para realização da síntese lógica de blocos combinacionais. CF usa uma abordagem ascendente para sintetizar funções Booleanas, sendo capaz de avaliar os custos das funções intermediárias e explorando dessa forma um grande número de combinações diferentes de funções candidatas. Há vantagens interessantes quando comparado à abordagem descendente da decomposição funcional. CF apresenta grande flexibilidade para criar algoritmos com resultados ótimos ou subótimos para diferentes aplicações. A estratégia proposta apresenta bons resultados para síntese de funções Booleanas visando diferentes tecnologias. CF é baseado nos seguintes princípios: (1) representação de funções lógicas como um par ligado com representações funcional e estrutural; (2) o algoritmo começa de um conjunto de funções iniciais; (3) funções mais simples são associadas para criar funções mais complexas; (4) existe uma ordem parcial que permite o uso da programação dinâmica; (5) um conjunto de funções permitidas pode ser mantido para reduzir o tempo de execução/consumo de memória. Este trabalho apresenta algoritmos de composição funcional para fatoração Booleana, incluindo fatoração ótima, fatoração considerando o operador OU-exclusivo, computação de cadeias mínimas de decisão e síntese de funções considerando somente portas lógicas majoritárias e inversores.

Palavras-Chave: Função Booleana, síntese lógica, composição funcional, cadeia mínima de decisão, fatoração Booleana, OU-exclusivo, porta majoritária.

1 INTRODUCTION

The increasing availability of digital technologies is adding new possibilities for investigative research and the way researchers work. Advances in many (if not most) areas of study depend now on the generation and manipulation of digital data, often in huge quantities.

Since Jack Kilby's invention of the first integrated circuit (IC), in 1958 (KILBY, 1959), unprecedented technological advances occurred, mainly in the electronic industry. In 1965, Intel Corp. co-founder, Gordon E. Moore, predicted a major miniaturization trend for the semiconductor industry, known as Moore's Law (MOORE, 1965). Moore's Law predicts that the number of available transistors being packed into a single IC would grow exponentially, doubling approximately every two years. This trend has been observed for more than four decades, and perhaps will continue for another decade or even longer, mainly for digital systems.

A digital system can often be divided into two portions: datapath and control logic. The datapath logic concerns with data computation and storage, and often comprises of structures such arithmetic logic units, buses, registers, regular memories (WAGNER; REIS; RIBAS, 2006). Datapath circuits are often composed of regular structures. Control logic is concerned with the control of these data processing units. Datapath and control logic can be very different in nature and the design of datapath and control logic uses different algorithms. It is common to have synthesis algorithms dedicated either to control logic or to datapath. Algorithms can also vary according to target implementation. The goal of this work is to provide an algorithm for logic synthesis that has enough flexibility to be adapted to control logic or datapath logic designed for different target implementations. The proposed approach is focused mainly on logic synthesis step, which will be described in the following.

1.1 Logic Synthesis

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 using a Register Transfer Level (RTL) description, where the design implementation regarding logic components is a logic gate netlist. Logic synthesis can be viewed as an essential step bridging high-level synthesis (which outputs an RTL description) and physical design (which takes as input a logic gate netlist). Logic synthesis involves the abstraction, representation, manipulation, transformation, analysis, and optimization of logic circuits. All these operations take place in the transformation from the RTL description to the gate level description. Logic synthesis performs automatic generation of logic gates netlists, such that timing

constraints imposed by the designer are respected, and the final logic gate netlist has minimum area and minimum power.

The main mathematical foundation of logic synthesis is the intersection of logic and algebra. The “algebra of logic” created by George Boole, in 1847, also known as Boolean algebra, is the core of logic synthesis. One of the most significant contributions connecting Boolean algebra and circuit design is Claude E. Shannon's M.Sc. thesis, “A Symbolic Analysis of Relay and Switching Circuits”, completed at the Massachusetts Institute of Technology (MIT), in 1937 (SHANNON, 1938). Shannon showed that the design and analysis of switching circuits could be formalized using Boolean algebra and switching circuits can be used to solve Boolean algebra problems.

The synthesis of digital circuits involves different views and levels of abstractions. Views can be associated with domains, as it is for the Y-chart proposed by Gajsky (GAJSKI; KHUN, 1983). Gajski divides circuit views into three types or domains: behavioral, structural and physical. Behavioral views express the expected behavior of a given circuit. Structural views express circuits as a list of interconnected sub-components. Physical views express how structural views are implemented physically. Generally speaking, the goal of any synthesis process is to add detail to existing design views. Synthesis is made to translate designs from the behavioral domain (views) to the structural domain, or from structural domain to physical domain. High-level synthesis does a translation from behavioral to the structural domain, where functional algorithms are mapped to structural representations composed of blocks such as arithmetic logics units, processors and read-only memories (ROMs). Logic synthesis starts from an abstract behavioral description, possibly at the register-transfer level (RTL), which is transformed into a structural netlist of logic gate networks. Logic synthesis is known to perform well to minimize and simplify control logic, which is irregular by nature. Consequently, logic synthesis is particularly useful for control-dominating applications like protocol processing. However, for more regular logic, like arithmetic-intensive applications, signal processing special techniques are used (SONG; PERKOSWI, 1998; SASAO, 2005; SASAO ET AL, 1995).

Logic synthesis is composed of technology independent and technology dependent steps, as shown in Figure 1.1. The initial RTL description is parsed into a technology-independent description. The initial parsing can include a control/data flow analysis that treats arithmetic/datapath independently of control flow. The arithmetic part can be synthesized by choice of possible pre-defined structures. The control flow logic is then treated with technology independent logic optimization algorithms. The initial gate-level implementation of the control flow portion is composed of generic logic gates (e.g. AND, OR and NOT gates), with no relationship to any specific technology. As a result, the structure at this point is a technology independent netlist (i.e., a list of gates connected by nets), and can be implemented in any technology using technology mapping. However, before the technology mapping process, a number of technology independent optimizations can be done to the technology independent netlist by logic restructuring techniques (MISHCHENKO, 2006). Once the technology mapping has been performed, it is followed by technology dependent optimizations and the insertion of logic to support design for testability (DFT). The physical synthesis begins after these steps (MICHELLI, 1994). A logic synthesis flow is illustrated in Figure 1.1.

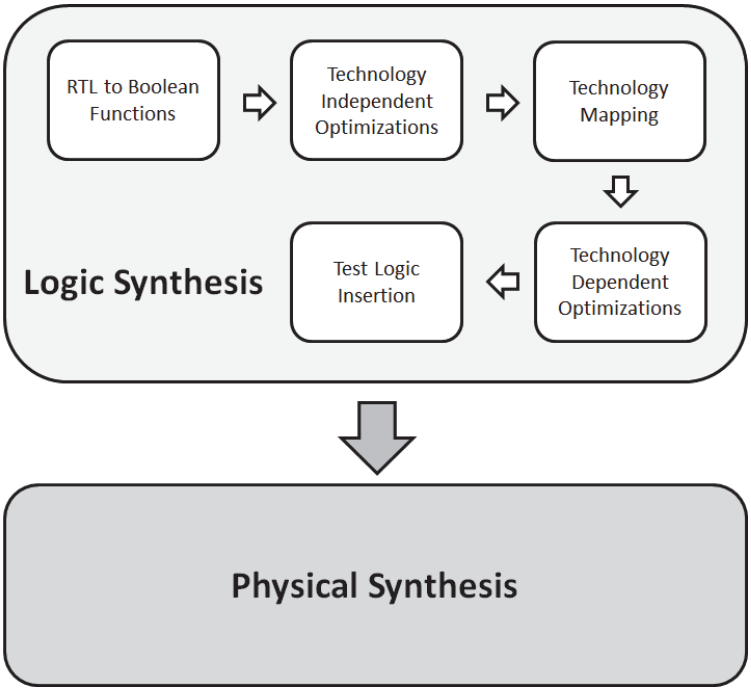


Figure 1.1: Logic synthesis flow.

1.2 Motivation and Challenges

Observing the logic synthesis field of study, it is easy to see many difficulties and problems with no definitive solution from a research point-of-view. There are many discussions on: "library based (LIU, 2011) versus library free (MARQUES ET AL, 2007)" approaches in mapping; which data structure to use for a given problem (SWORDS; HUNT, 2010; MISHCHENKO; CHATTERJEE; BRAYTON, 2006); and when to use complex gates (KONG; HUSSAIN; OVERHAUSER, 1997) or simple gates (MARKOV; SYLVESTER; BLAAUW, 2008). There are many objectives comprising minimum area, minimum power, maximum performance or any combination of these. Most logic synthesis algorithms are computationally hard problems, and solutions demonstrate approaches based on theoretical studies but also empirical solutions.

There are many subareas in logic synthesis. Some examples are factoring algorithms, majority based circuit synthesis and functional decomposition. Many of logic synthesis methods (if not all) may rely on the exploitation of Boolean functions properties. As a consequence, fast methods to compute Boolean function properties are needed to allow the use of these properties in the logic synthesis flow. One of these properties is the minimum decision chain (MDC) (MARTINS ET AL, 2011a), (MARTINS ET AL, 2011b). The MDC of a function f is related to the minimum worst case number of series switches required to implement a switch network for the function f in a single stage. The top-down algorithm for MDC computation (SCHNEIDER ET AL, 2005) is slow in some cases of interest (up to 5 transistors in series), so limiting the MDC use in practice.

Factoring is another important procedure for logic synthesis tools. It consists in the conversion of a logic function into a logically equivalent parenthesized expression or

factored form (BRAYTON, 1987). This factored form in general presents a reduced implementation cost. There are other costs associated with a factored form, but the state-of-art algorithms only concern to obtain a minimal factored form, not exploring other costs. Besides that, the state-of-art algorithms do not synthesize expressions with the exclusive-OR operator, which is interesting for arithmetic and testing circuits.

On the other hand, the research in majority logic synthesis dates back to 1960s, when majority logic circuits were called threshold circuits (MUROGA, 1971). A threshold circuit uses majority gates with weighted inputs, having applications mainly in analog circuits. With the CMOS technology achieving the scaling limits, there are many candidate technologies to replace the CMOS, such as tunneling phase logic (TPL), single electron tunneling (SET) and quantum cellular automata (QCA). All these mentioned technologies use majority or minority gates as primitive elements in the circuit (ZHANG; GUPTA; JHA, 2005). Unfortunately, there are no algorithms capable of generating optimal synthesis (considering the number of majority gates as costs) for functions with more than 3 inputs.

Functional decomposition (FD) is a method for combinational logic synthesis in which a Boolean function is decomposed into a set of smaller functions that implement the Boolean function. FD has been introduced by the pioneering works of (ASHENHURST, 1959) and (CURTIS, 1962). The results of functional decomposition are in the functional domain, meaning that it can produce non-trivial logic rewritings that are very suitable to overcome the structural bias. A logic synthesis algorithm with structural bias has the results strongly dependent on the algorithm input data, and this is not a desired characteristic. FD has been extensively used for FPGA mapping, as it is simple to control the number of inputs of each sub-function. However, FD has two critical drawbacks in this context. Firstly, it is a top-down approach, which breaks the original function to be decomposed into smaller ones, so the implementation cost of these functions is not necessarily known. Secondly, FD involves costly operations for one possible decomposition, relying on complex operations as counting the number of distinct subfunctions, test inversions, and so on.

All these drawbacks discussed in FD, factorization, MDC computation, and majority gate circuit synthesis can be overcome if a bottom-up approach is used since the costs of initial functions are known, the logic operations are simple, the subfunctions have sub-optimal and optimal implementations, and a control cost can be easily set.

1.3 Objectives

This thesis proposes a novel technique based on functional composition (FC) to overcome the drawbacks of functional decomposition applied to local function rewriting. It is a novel synthesis paradigm that performs a bottom-up association of Boolean functions as 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 enables dynamic programming is respected; (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 due to the use of a bottom-up approach, where the implementation costs of all subfunctions generated during the synthesis process are known. In this thesis, three different FC algorithms are exemplified. All three algorithms are obtained by configuring FC for various purposes.

The first application is an approach to efficiently compute MDC, especially from the cases of interest. The second application is a Boolean factoring algorithm that is capable of factorizing expressions considering one or more costs besides the expression size. A second Boolean factoring algorithm has the capability to factorize expressions using the exclusive-OR operator. The third application is a majority gate circuit synthesis algorithm superior to the existing ones.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 provides some basic concepts useful for a better understanding of the reader. Chapter 3 explains the paradigm of FC, describing the general principles, which are used in all the applications proposed herein. Chapter 4 explains the MDC computation using FC. Chapter 5 and Chapter 6 propose two novel FC-based Boolean factoring algorithms which provide important improvements over previously available factoring procedures. Chapter 7 explores the characteristics of FC to synthesize circuits containing majority gates and inverters. The last chapter presents the conclusion of the thesis, summarizes the contributions of this work, summarizes the major contributions, and outlines some possible future works.

2 BOOLEAN LOGIC CONCEPTS

In this chapter, important concepts of logic synthesis that are necessary for the complete understanding of this thesis are reviewed. The concepts described are Boolean algebra; Boolean equations representation forms; representation of Boolean functions in a data structure format; the computing properties of Boolean functions and the logic design using only AND-eXclusive-OR (AND-XOR) expressions. Readers with knowledge in this field can skip this chapter without compromising the understanding of the contents discussed in the next chapters.

2.1 Boolean Functions

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

$$f : B^n \mapsto B$$

where $x = [x_1,\ldots,x_n] \in B^n$ is the input of f. This is a representation of a completely specified Boolean function (CSF) taking values from 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 $(X^{ON} \subseteq B^n)$ is the set of input values x such that $f(x) = 1$, and the off-set $(X^{OFF} \subseteq B^n)$ is the set of input values x such that $f(x) = 0$.

2.2 Canonical Representations of Boolean Functions

A canonical representation of a Boolean function means that for each possible function f, there is only a unique representation, considering a given representation type. In this section, two canonical representations are presented: truth tables and binary decision diagrams (BDD).

2.2.1 Truth Table

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 $f : x \mapsto y \mid y \in B$, where the values of $x = [x_1, x_2, x_3]$ is indicated in Table 2.1.

Table 2.1: Truth table for a logic function with 3 variables

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

For this function, $X^{ON} = \{\{[0,0,0],\ [0,0,1],\ [1,0,0],\ [1,0,1],\ [1,1,0]\}$ and $X^{OFF} = \{[0,1,0],\ [0,1,1],\ [1,1,1]\}$.

The assignment of a Boolean function can be exhaustively enumerated with a truth table, where every truth assignment has a corresponding y value.

Truth tables are canonical representations of Boolean functions. That is, two Boolean functions are equivalent if and only if they have the same truth table. The canonicity of Boolean data structures is an important property because is simple to check the equality between two Boolean functions.

Because of exhaustive enumeration, it is not feasible to represent a Boolean function through a truth table containing many input variables, e.g. a number of inputs greater than 20. The truth table has space complexity of $O(2^n)$ since this is the number of lines in a truth table. Hence, it is not memory feasible to represent many functions with 2^{20} bits per function, for instance.

By storing the truth table data as a computer word or an array of words, basic Boolean operations can be done in constant time by parallel operation over their truth tables. For example, the y output of the truth table shown in Table 2.1 can be represented as 01110011_2 (in binary format) or 73_{16} (in hexadecimal format), where the most significant bit is the leftmost digit.

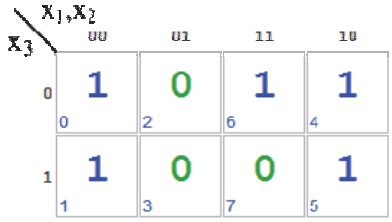


Figure 2.1: Karnaugh map representation of the function shown in Table 2.1.

The Karnaugh map is another way of representing the information of a function (KARNAUGH, 1953). In a Karnaugh map, the Boolean variables are ordered according to the principles of Gray code in which only one variable changes in between adjacent

neighboring values (GRAY, 1947). It is designed to present the information in a bi-dimensional way that allows easy grouping of neighboring terms. A Karnaugh map representation for the function seen in Table 2.1 is shown in Figure 2.1.

2.2.2 Binary Decision Diagrams

A Boolean function can be represented as a rooted, directed, acyclic graph which consists of decision nodes and two terminal nodes called 0-terminal and 1-terminal, also known as BDDs. BDDs were first proposed by (LEE,1959), and further developed by Akers (AKERS, 1978). To reduce the number of decision nodes in the representation, (BRYANT, 1986) proposed several reduction rules, leading to the well-known Reduced Ordered BDDs (ROBDDs).

The BDD from Akers is a directed graph *G(V, E)*. All vertices $v \in V$, except for the root and the leaf vertices, have one edge ingoing to them and two outgoing edges from them, where an edge $e \in E$. The two outgoing edges from a vertex point to the children vertices are called low and high vertices, and are denoted by $\eta(v)$ and $\lambda(v)$, respectively. Moreover, each vertex has an attribute called variable and is denoted by $\phi(v)$. When $\phi(v) = 0$, the vertex $\eta(v)$ is chosen and when $\phi(v) = 1$ the vertex $\lambda(v)$ is chosen. The root vertex does not have an edge incident to it. The leaf vertices do not have any edges leaving from them. The vertex is shown in Figure 2.2. A BDD example is illustrated in Figure 2.3, where the dashed line is the $\eta(v)$ and the full line is $\lambda(v)$.

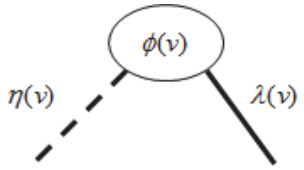


Figure 2.2: Vertex representation for a BDD node.

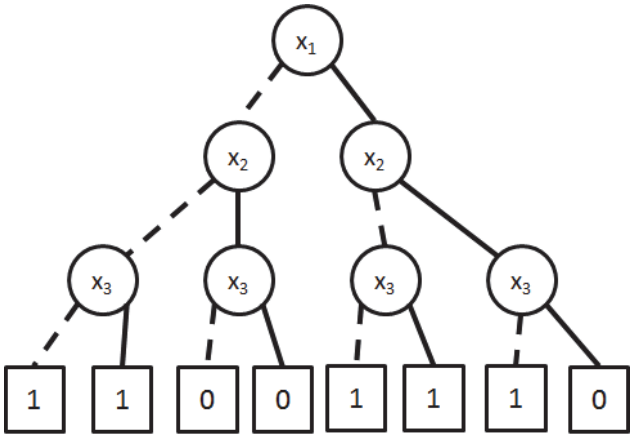


Figure 2.3: BDD representation from the function in Table 2.1.

An Ordered BDD (OBDD) is a BDD with the nodes on every path from the root node to a terminal node of the BDD follow the same variable ordering. An OBDD can be reduced to an ROBDD. The derived ROBDD has the smallest number of nodes under a given variable ordering.

ROBDDs are structurally (and also functionally) isomorphic, this means that two functional equivalent ROBDDs have the same graph structure, considering the same

variable ordering for two ROBDDs. Every function has only one ROBDD for a given variable ordering, being a canonical representation of its respective Boolean function. The ROBDD obtained from OBDD of Figure 2.3 is shown in Figure 2.4.

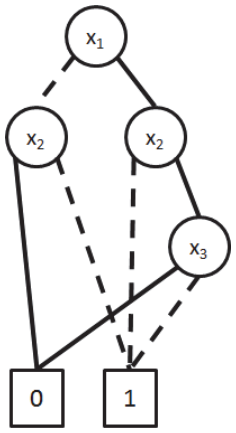


Figure 2.4: ROBDD from the OBDD in Figure 2.3.

The size of the ROBDD is determined both by the function being represented and the chosen ordering of the variables. There exist Boolean functions that depend on the ordering of the variables, these functions would end up being represented by a graph whose number of nodes would be linear on the number of Boolean variables in the best case and exponential at the worst case. The variable ordering chosen in the BDD of Figure 2.3 was [x_1,x_2,x_3]. Choosing the variable ordering [x_2,x_1,x_3] results in the ROBDD shown in Figure 2.5. This ROBDD has one node less than the ROBDD from Figure 2.4, hence decreasing the memory consumption.

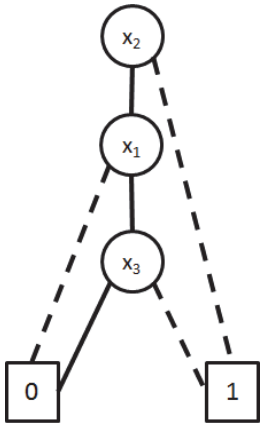


Figure 2.5: ROBDD with the same function from Table 2.1, but with different variable ordering.

The advantage of ROBDD is the possibility to perform Boolean operations directly on the compressed representation, i.e., without decompressing an ROBDD to an OBDD to do the operation.

2.3 Boolean Operations

Four basic Boolean operations are discussed in this section. These operations are illustrated in Figure 2.6, and the truth tables are representing the operations are shown in Table 2.2.

The complement or negation (NOT, $\bar{}$) 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*, $h = g \cdot f$ 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*, $i = g + f$ is defined to be the logic function *i*, where $i_{on} = f_{on} \cup g_{on}$.

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

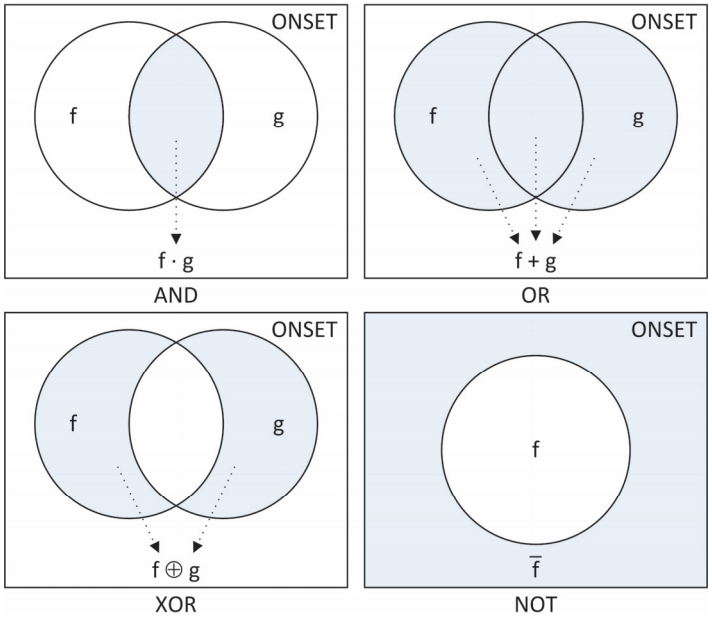


Figure 2.6: Boolean operations: the blue area represents the result of each operation.

Table 2.2: 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* *j*
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

2.4 Properties of Boolean Functions

There are some important Boolean properties, described in the following subsections.

2.4.1 Shannon Expansion and Cofactors

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

$$f(x_1,\ldots,x_i,\ldots x_n) = x_i \cdot f(x_1,\ldots,1,\ldots x_n) + \overline{x_i} \cdot f(x_1,\ldots,0,\ldots x_n) \tag{2.1}$$

The cofactor is a sub-element of a Shannon expansion. The Shannon expansion is a way to express a Boolean function by the sum 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:

$$f_{x_i} = \{f(x_1,\ldots,x_i,\ldots x_n) \mid x_i = k, k \in B\} \tag{2.2}$$

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 x_i 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.

There is also a Shannon expansion representation using the exclusive-OR operator:

$$x_i \cdot f(x_1,\ldots,1,\ldots x_n) \oplus \overline{x_i} \cdot f(x_1,\ldots,0,\ldots x_n) \tag{2.3}$$

This expansion is used to generate expressions using only AND-XOR operators.

2.4.2 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 $U(f,x_k)$ denote the unateness detection function of a variable x_k in the function *f*, and auxiliary function $i = f_{x_k=1} + f_{x_k=0}$, we have:

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

For instance, consider the function *f* as*:*

$$f = (x_1 + \overline{x_2}) \cdot (x_3 + \overline{x_1} \cdot x_4)$$

The cofactor $f_{x_1=1} = x_3$ and $f_{x_1=0} = \overline{x_2} \cdot (x_3 + x_4)$. Generating the auxiliary function $i = x_3 + (\overline{x_2} \cdot x_4)$, it is possible to see that the variable x_1 is binate in f. Repeating the process for the other variables in f, x_3, x_4 are positive unate and x_2 is negative unate.

2.4.3 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 $O(f,g)$ denote the order of f against g, and h be the auxiliary function $h = f + g$, we have:

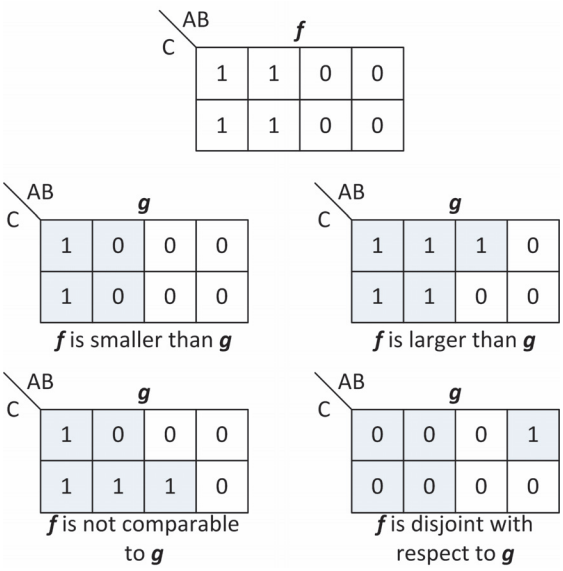


Figure 2.7: Order visualized in the Karnaugh map.

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

The order of two functions can be easily observed in a Karnaugh map, shown in Figure 2.7. The hachured area represents the auxiliary function h for each function g.

2.4.4 Symmetry

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 $f(x_1, \ldots x_n)$ 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 $S(f, x_i, x_j)$ denote the symmetry check of variables x_i and x_j in the function f, g and h auxiliary functions, we have:

$$g = f_{x_i=1,x_j=0}$$
$$h = f_{x_i=0,x_j=1}$$

$$S(f,x_i,x_j) = \begin{cases} symmetric & g = h \\ not\ symmetric & otherwise \end{cases}$$

The antisymmetric property is similar, changing only the cube cofactors to be checked. Let $AS(f,x_i,x_j)$ denote the antisymmetry check of variables x_i and x_j in the function f, and k and m auxiliary functions, we have:

$$k = f_{x_i=0,x_j=0}$$
$$m = f_{x_i=1,x_j=1}$$

$$AS(x_i,x_j) = \begin{cases} antisymmetric & k = m \\ not\ \text{anti}symmetric & otherwise \end{cases}$$

Example: Consider the function f as:

$$f = x_1 \cdot x_2 + x_1 \cdot x_3 + x_2 \cdot x_3$$

x_1, x_2, x_3 are symmetric.

Example 2: Consider the function g as:

$$g = x_1 \cdot \overline{x_2} + \overline{x_1} \cdot x_2$$

x_1, x_2 are symmetric and antisymmetric.

2.4.5 Davio Expansion

There are other ways to expand a Boolean expression with the exclusive-OR operator. The Reed-Muller or Davio expansion is one of the most used expansions with this objective (THAYSE, 1973). Consider the Boolean derivation of f as follows:

$$\frac{\partial f}{\partial x_i} = f_{x_i=1} \oplus f_{x_i=0} \tag{2.3}$$

The positive Davio expansion is:

$$f = f_{x_i=0} \oplus x_i \frac{\partial f}{\partial x_i} \tag{2.4}$$

The negative Davio expansion is:

$$f = f_{x_i=1} \oplus \overline{x_i} \frac{\partial f}{\partial x_i} \tag{2.5}$$

For instance, consider the function $f = x_1 \cdot x_2 + x_3$. Applying the positive Davio expansion in the variable x_1, the result is indicated in Equation 2.6.

$$f = \begin{array}{l} x_3 \oplus \overline{x_1} \cdot (x_2 \cdot x_3 \oplus x_3) \\ x_3 \oplus \overline{x_1} \cdot x_2 \cdot x_3 \oplus \overline{x_1} \cdot x_3 \end{array} \quad (2.6)$$

2.4.6 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 2.3 is presented a self-dual function, called *f*.

Table 2.3: 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 it characterizes a self-dual function.

2.5 Boolean Equations

An algebraic representation of *f* is a Boolean expression that evaluates to 1 for all inputs in X^{ON}, and evaluates to 0 for all inputs in 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 the operator $\cdot$) of the *n* input variables $[x_1, \ldots, x_j, \ldots, x_n]$, the variable x_j appears complemented if the corresponding value of the input variable in the row of the truth table is 0 and uncomplemented 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 2.3, and applying the rules above, the Equation 2.7 is obtained.

$$f = \overline{x_1} \cdot x_2 \cdot x_3 + x_1 \cdot \overline{x_2} \cdot x_3 + x_1 \cdot x_2 \cdot \overline{x_3} + x_1 \cdot x_2 \cdot x_3 \quad (2.7)$$

2.5.1 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 $[x_1, x_2, x_3, x_4]$, being x_1 a positive literal and $\overline{x_2}$ a negative literal.

2.5.2 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 product, being called product-of-sum (POS).

2.5.2.1Minterms 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 uncomplemented 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 uncomplemented 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 uncomplemented form) is called a maxterm.

For example, consider a function with 3 variables with input assignment [x_1,x_2,x_3]. 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 $x_1 + x_2 + \overline{x_3}$), 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 $x_1 + \overline{x_2} + x_3$) with input assignment [1,0,1], and m_7 is $x_1 \cdot x_2 \cdot x_3$ (M_7 is $x_1 + x_2 + x_3$) with input assignment [1,1,1].

2.5.2.2Implicants, 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.

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 $f = \overline{x_1} \cdot \overline{x_3} + x_1 \cdot x_3 + \overline{x_2} \cdot \overline{x_3}$, the Karnaugh map is shown in Figure 2.8, and the covering table of *f* is shown in Figure 2.9.

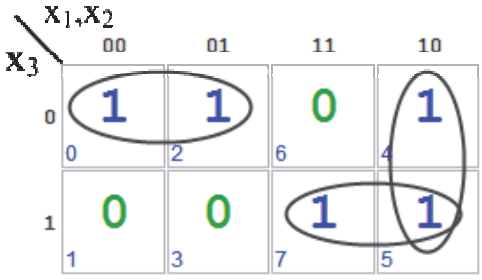


Figure 2.8: Karnaugh map of function *f*.

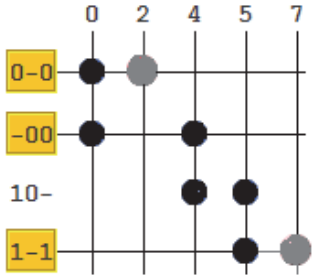


Figure 2.9: Covering table of function *f*.

The columns in Figure 2.9 represent the minterms, and the lines represent the implicants. The following symbols represent each prime implicant: '**0**' – the respective variable is complemented; '**1**' – the respective variable is uncomplemented, and '**–**' (dash) – the respective variable is absent (does not care). A *don't care* variable can assume the value **0** or **1**. For example, the prime implicant [0-0] represents the implicants [000] and [010].

The two gray spheres represent implicants that are only covered by one prime implicant, and the black spheres represent the minterms covered by the respective cube. The [0-0] and [1-1] are essential prime implicants. The [-00] and the [10-] are prime implicants, but not essential prime implicants, since the minterm 4 is covered by both. An **irredundant sum-of-products** (ISOP) is a SOP where no product can be deleted without changing the function. The **irredundant product-of-sums** (IPOS) is a POS where no sum can be deleted without changing the function.

2.5.3 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 2.8. The Equation 2.8 can be factored in a more compact, parenthesized representation represented by Equation 2.9.

$$f = b \cdot d \cdot e \cdot f + b \cdot c + a \cdot d \cdot e \cdot f + a \cdot c \tag{2.8}$$

$$f = (a + b) \cdot (c + d \cdot (e \cdot f)) \tag{2.9}$$

The logic tree of the two level expressions and the factored expressions are shown in Figure 2.10 and Figure 2.11, respectively. Note that the logic tree of the factored expression has three levels of Boolean operations. The number of literals is also reduced, from 12 literals in the SOP expression to 6 literals in the factored expression.

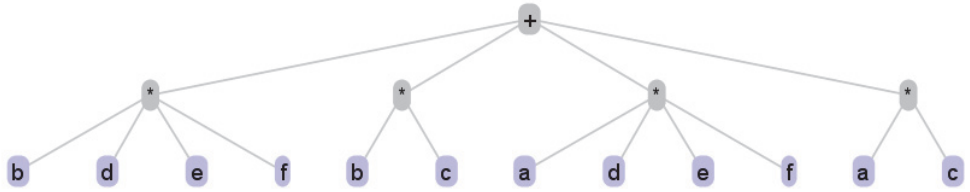


Figure 2.10: Logic tree of the two level expression representation of the expression $b \cdot d \cdot e \cdot f + b \cdot c + a \cdot d \cdot e \cdot f + a \cdot c$.

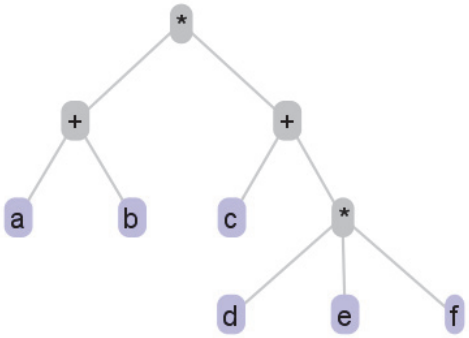


Figure 2.11: Logic tree of the factored expression representation of the expression $(a + b) \cdot (c + d \cdot (e \cdot f))$.

2.5.3.1Read-Once Functions

A function f is called read-once if it can be represented by an expression where each variable appears no more than once. For instance, the factored expression $(a + b) \cdot (c + d \cdot (e \cdot f))$ is read-once.

2.6 Comparison Between Boolean Functions Representations

This chapter presented three ways of describing Boolean functions. Table 2.4 summarizes the essential characteristics of each kind of representation.

Table 2.4: Characteristics of the representations forms of Boolean functions.

	Truth Table	BDD (ROBDD)	Expression
Size	Exponential	Linear to Exponential	Linear
Boolean Operation	Linear	Linear to Exponential	Constant
Equality	Linear	Constant	Exponential
Canonicity	Yes	Yes	No

2.7 Two Level AND-XOR Expressions

Most of the logic design methods are based on AND-OR-NOT expressions. However, the use of XOR has particular interest in arithmetic and telecommunication circuits, reducing the complexity of switching networks. There are many forms of expression representation, generated by using Davio expansion, as discussed below.

- **Positive Polarity Reed-Muller Expression (PPRM)**

PPRM is obtained by expanding an expression recursively using the positive Davio expansion. Negative polarity variables can be represented as $\overline{x} = x \oplus 1$. The resulting expression is canonical.

Example: The Equation 2.10 is a representation of $\overline{x_1} \cdot \overline{x_2}$ as a PPRM.

$$\overline{x_1} \cdot \overline{x_2} = (x_1 \oplus 1) \cdot (x_2 \oplus 1) = x_1 \cdot x_2 \oplus x_1 \oplus x_2 \oplus 1 \tag{2.10}$$

- **Fixed Polarity Reed-Muller Expression (FPRM)**

FPRM representation is a generalization of PPRM. Each variable can be expanded using the positive or negative Davio expansion, but not both at the same time.

Example: Representation of $x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}$ as an FPRM, using a negative Davio expansion in x_1 and a positive Davio expansion in x_2 and x_3.

Using the property $x \cdot y = 0 \Leftrightarrow x + y = x \oplus y$:

$$x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} = x_1 \cdot x_2 \cdot x_3 \oplus \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}$$

$$x_1 \cdot \overline{x_2} \cdot \overline{x_3} \oplus \overline{x_1} \cdot x_2 \cdot x_3 = \quad x_1 \cdot (x_2 \oplus 1)\overline{x_3} \oplus (x_1 \oplus 1) \cdot x_2 \cdot (\overline{x_3} \oplus 1)$$
$$x_1 \cdot \overline{x_3} \oplus x_1 \cdot x_2 \oplus x_2 \cdot \overline{x_3} \oplus \cdot x_2$$

- **Kronecker Expression (KRO)**

KRO representation is a generalization of FPRM. Each function can be expanded using the positive or negative Davio expansion or the Shannon expansion for all variables.

Example: Representation of $x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}$ as a KRO, using a Shannon expansion in x_1, x_2 and x_3:

$$\begin{aligned} x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} = \quad & x_1(x_2 \cdot x_3) \oplus \overline{x_1}(\overline{x_2} \cdot \overline{x_3}) \\ & x_1 \cdot x_2 \cdot (x_1 \cdot x_3) \oplus \overline{x_1} \cdot \overline{x_2} \cdot (\overline{x_1} \cdot \overline{x_3}) \\ & x_1 \cdot x_2 \cdot x_3 \cdot (x_1 \cdot x_2) \oplus \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}(\overline{x_1} \cdot \overline{x_2}) \\ & x_1 \cdot x_2 \cdot x_3 \oplus \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \end{aligned}$$

- **Pseudo Reed-Muller Expression (PSDRM)**

PSDRM representation is another generalization of an FPRM. For each expansion using positive or negative Davio expansion in an expression, there are two resulting subfunctions. For each of the two subfunctions, they can be expanded using positive or negative Davio expansions, using different expansions for each subfunction.

Example: Representation of $x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}$ as a PSDRM, using a positive Davio expansion in x_1.

$$x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} = \overline{x_2} \cdot \overline{x_3} \oplus x_1(x_2 \cdot x_3 \oplus \overline{x_2} \cdot \overline{x_3})$$

The subexpression $\overline{x_2} \cdot \overline{x_3}$ is a PSDRM since all variables are expanded with the negative Davio expansion. Expanding $x_2 \cdot x_3 \oplus \overline{x_2} \cdot \overline{x_3}$:

$$\begin{aligned} x_2 \cdot x_3 \oplus \overline{x_2} \cdot \overline{x_3} = \quad & \overline{x_3} \oplus x_2 \cdot (x_3 \oplus \overline{x_3}) \\ & \overline{x_3} \oplus x_2 \end{aligned}$$

The final expression is:

$$\begin{aligned} x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} = \quad & \overline{x_2} \cdot \overline{x_3} \oplus x_1 \cdot (\overline{x_3} \oplus x_2) \\ & \overline{x_2} \cdot \overline{x_3} \oplus x_1 \cdot \overline{x_3} \oplus x_1 \cdot x_2 \end{aligned}$$

- **Pseudo Kronecker Expression (PSDKRO)**

PSDKRO representation is the generalization of KRO representation. For each expansion using positive or negative Davio expansion or the Shannon expansion in an expression, there are new subfunctions. For each of the new subfunctions, they can be expanded using positive or negative Davio expansions or the Shannon expansion, using different expansions for each subfunction.

Example: Representation of $x_1 + x_2 \cdot x_3$ as a PSDKRO, using a Shannon expansion in x_1, $x_1 \oplus \overline{x_1} \cdot x_2 \cdot x_3$ is obtained:

Decomposing $x_2 \cdot x_3$ using the negative Davio expansion in x_2, is obtained $x_3 \oplus \overline{x_2} \cdot x_3$.

Decomposing $x_3 \oplus \overline{x_2} \cdot x_3$ using the positive Davio expansion in x_3, is obtained $x_3 \oplus \overline{x_2} \cdot x_3$.

Applying all the expansions:

$$\begin{aligned} x_1 + x_2 \cdot x_3 = \quad & x_1 \oplus \overline{x_1} \cdot x_2 \cdot x_3 \\ & x_1 \oplus \overline{x_1} \cdot (x_3 \oplus \overline{x_2} \cdot x_3) \\ & x_1 \oplus \overline{x_1} \cdot x_3 \oplus \overline{x_1} \cdot \overline{x_2} \cdot x_3 \end{aligned}$$

- **Generalized Reed-Muller Expression (GRM)**

GRM representation is the generalization of a PPRM expression. Each literal at each product can have arbitrary polarity.

Example: The expression of $x_1 \oplus x_2 \oplus \overline{x_1} \cdot \overline{x_2}$ is a GRM but not a PSDKRO since this expression cannot be achieved using positive/negative Davio or Shannon expression.

- **Exclusive-OR Sum-of-Products Expression (ESOP)**

ESOP representation is a generalization of any AND-EXOR expression. An ESOP is a logic expression that combines arbitrary product terms using XORs.

Example: The expression of $x_1 \oplus x_2 \oplus x_1 \cdot x_2 \oplus \overline{x_1} \cdot \overline{x_2}$ is an ESOP. This representation cannot be achieved by a PSDKRO or GRM.

Figure 2.12 summarizes all classes of AND-EXOR expressions in sets, and the characteristics of all classes are compiled in Table 2.5.

Table 2.5: Characteristics of the AND-XOR expressions.

	Positive Davio expansion	Negative Davio expansion	Shannon expansion	Different expansions	Arbitrary polarity	Arbitrary products
PPRM	✓					
FPRM	✓	✓				
KRO	✓	✓	✓			
PSDRM	✓	✓		✓		
PSDKRO	✓	✓	✓	✓		
GRM	✓	✓		✓	✓	
ESOP	✓	✓	✓	✓	✓	✓

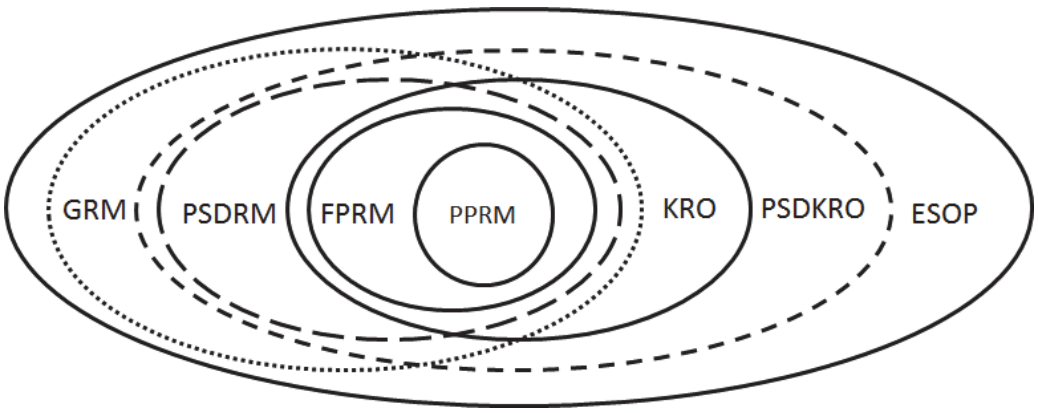


Figure 2.12: Relationship between various classes of AND-EXOR expressions.

3 GENERAL PRINCIPLES OF FUNCTIONAL COMPOSITION

This thesis proposes the functional composition aiming at overcoming the drawbacks of functional decomposition applied to local function rewriting. FC is a novel synthesis paradigm that performs bottom-up association of Boolean functions as opposed to the top-down functional decomposition approach. By performing a bottom-up approach, the costs of initial functions are necessarily known, the logic operations are simple, the subfunctions have sub-optimal and optimal implementations, and a control cost can be easily set.

In this chapter, the functional composition (FC) paradigm is described, and the novelty of FC is introduced by comparing with functional decomposition (FD). The principles of functional composition are discussed.

3.1 Functional Decomposition

Functional decomposition (FD) is a method for combinational logic synthesis in which a Boolean function is decomposed into a set of subfunctions. FD has been introduced by the pioneering works of (ASHENHURST, 1959) and (CURTIS, 1962). The results of FD 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, 2005). FD has been extensively used in FPGA mapping since it is easy to control the number of inputs at each subfunction (STANION; SECHEN, 1995). There are many related works on functional decomposition, such as disjoint support decomposition (DSD) (BERTACCO; DAMIANI, 1997) and bidecomposition (YANG; CIESIELSKI, 2002).

The disjoint support decomposition of a Boolean function $F(x_1,\ldots,x_n)$ consists in representing f using 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 3.1.

In general, a function has several disjoint support decompositions, which can be superimposed to obtain decompositions with finer granularity. Moreover, it is possible to recursively search 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.

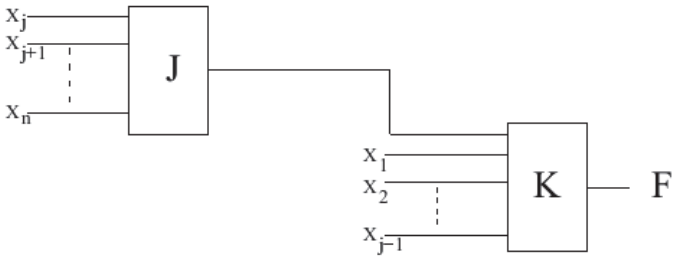


Figure 3.1: A disjoint support decomposition for *F*. Source: (PLAZA; BERTACCO, 2005.)

The bidecomposition is based on a binary decision diagram decomposition technique which supports decomposition structures of AND, OR, XOR, and complex MUX, both algebraic and Boolean, using the concept of 0-dominators and 1-dominators (KARPLUS, 1988), x-dominators and MUX decomposition. Karplus introduced the idea of a 1-dominator and 0-dominator and showed their relationship to algebraic AND/OR decomposition, illustrated in Figure 3.2. In a BDD without the complement edges, a 1-dominator (0-dominator) is a node which belongs to every path from the root to terminal node 1 (0). The x-dominator is a dominator that helps to identify algebraic XOR decomposition (YANG; CIESIELSKI, 2002).

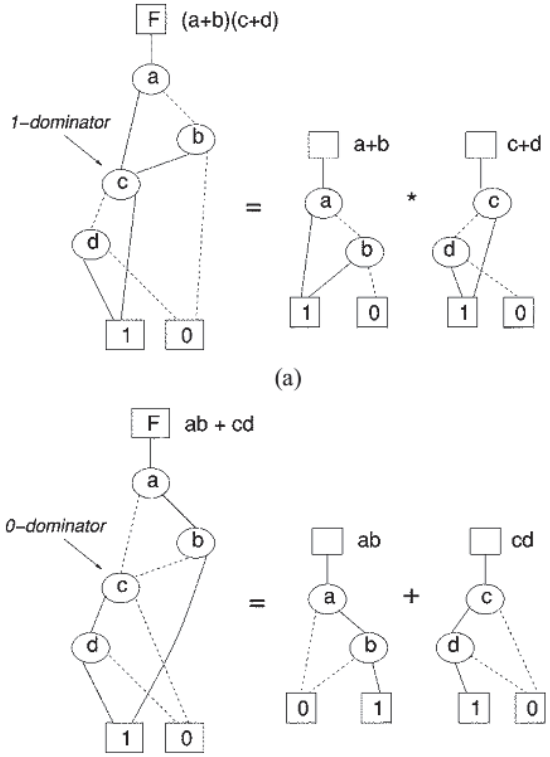


Figure 3.2: Algebraic decompositions of Karplus: (a) conjunction decomposition, *F=(a+b)(c+d)*, based on 1-dominator and (b) disjunctive decomposition, *F=ab+cd*, based on 0-dominator. Source: (YANG; CIESIELSKI, 2002).

The bidecomposition method is efficient in synthesizing both AND/OR and XOR-intensive functions. The algorithm makes a horizontal cut in a BDD and then the two

resulting BDDs will have disjoint support. There are conjunctive and disjunctive decompositions. In Figure 3.3, there is an example of conjunctive decomposition.

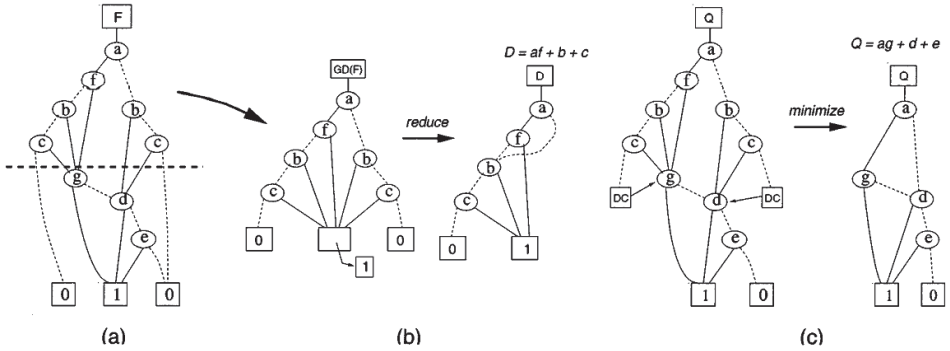


Figure 3.3: Conjunctive BDD decomposition: (a) original function *F*, (b) generalized dominator and Boolean divisor *D*, and (c) computing quotient *Q* from *F*. Source: YANG; CIESIELSKI, 2002.

In the last decade, logic synthesis has made a move forward relying on the representation of Boolean functions as integers, especially considering the rewriting of small portions of and-inverter graphs (AIGs) (MISHCHENKO, 2006). In this context, the task of rewriting a small Boolean function, without structural bias, has significantly greater value. FD can play a major role in this regard. However, the FD has two critical drawbacks in this context. First of all, it is a top-down approach, which decomposes the original function into smaller ones. Thus, the implementation cost of these functions is not necessarily known. Secondly, the FD depends on costly operations for one possible decomposition, relying on complex operations as count subfunctions extracted, test inversions, and so on.

3.2 General Principles of Functional Composition

The FC paradigm is based on some general principles (MARTINS; RIBAS; REIS, 2012a) (MARTINS; RIBAS; REIS, 2012b). These principles 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.

3.2.1 Bonded-Pair Representation

FC uses bonded-pairs to represent logic functions. The bonded-pair is a data structure that contains one functional and one structural representation of the same Boolean function. The functional representation is used to avoid the structural bias dependence, making FC a Boolean method. Usually the functional representation needs to be a canonical representation, as a truth table or an ROBDD structure. The structural representation used in bonded-pairs is related to the final implementation of the target function, controlling costs in such final solution. In principle, it is not a canonical implementation, as costs may vary. In Figure 3.4 is illustrated an example of a bonded-pair representation with structural part implemented as an expression and the functional

part as a truth table represented as an integer, considering the most significant bit the leftmost.

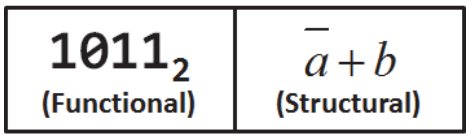


Figure 3.4: Bonded-Pair representation example.

3.2.2 Initial Functions

The FC paradigm computes new functions by associating known functions. As a consequence, a set of initial functions is necessary before starting the algorithm. The set of initial functions needs to have two characteristics: (1) the bonded-pairs for the initial functions are the initial input of any algorithm based on FC; (2) the initial functions must have known costs (preferable minimum costs) for each function, allowing the computation of the cost for derived functions. For instance, in Figure 3.5 is illustrated a possible set of initial functions with two variables, using the BP representation shown in Figure 3.4.

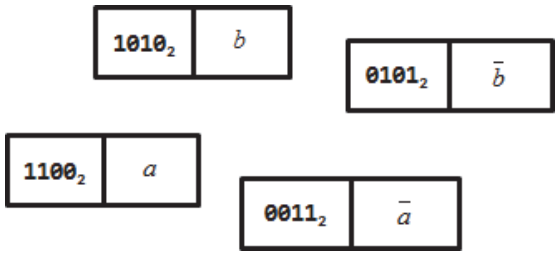


Figure 3.5: Example of initial bonded-pairs.

3.2.3 Bonded-Pair Association

When a logic operation (e.g. logic OR) is applied to bonded-pairs, the operation is 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 difficult and inefficient. The main advantage of the bonded-pair association is the operations occurring in the functional and structural domain in parallel, avoiding conversions. To maintain the control over the structural representation, avoiding the inefficient structures and a lack of control of converting one type into another, it is used the bonded-pair representation. Figure 3.6 presents the association of bonded-pairs. The bonded-pair $<F_3,S_3>$ is obtained from bonded-pairs $<F_1,S_1>$ and $<F_2,S_2>$. The computation of the functional part ($F_3=F_1+F_2$) is independent of the computation of the structural part ($S_3=S_1+S_2$).

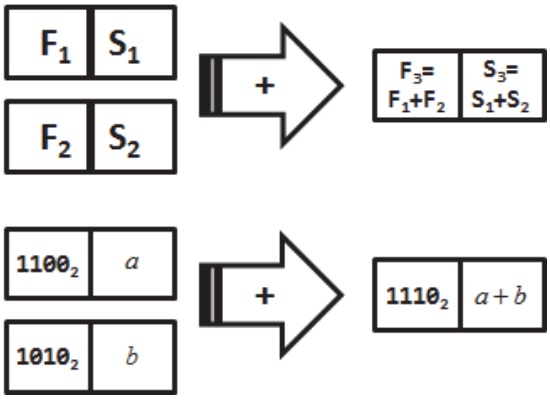


Figure 3.6: Bonded-pair association example.

3.2.4 Partial Order and Dynamic Programming

The key concept of dynamic programming (DP) 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 sub-structure. It starts by solving sub-problems and then combining the sub-problem solutions to obtain a complete solution. In functional composition, DP 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 BPs) 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 BP representation. This concept is illustrated in Figure 3.7.

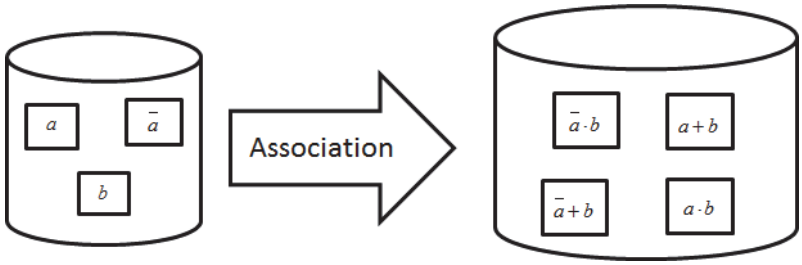


Figure 3.7: When combining the elements of a bucket, a new element is generated and stored in a new bucket.

3.2.5 Allowed Functions

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. For instance, this is the case

of read-once factoring. The read-once factoring algorithm will be discussed in Chapter 5.

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 3.8, based on the example shown in Figure 3.7. A heuristic algorithm discarded the function $a + b$, reducing the amount of functions inserted in the bucket.

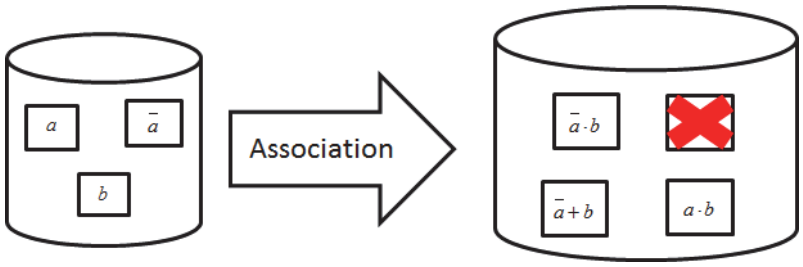


Figure 3.8: Some elements of Figure 3.7 can be removed to reduce the number of elements in a bucket, improving memory and execution time.

3.3 General Flow

In Figure 3.9, the general flow for algorithms following the FC principles is shown. The first step consists in parsing the target function. Then, the initial bonded-pairs are generated and compared with the target function. If the target function is not found, the allowed functions are computed and inserted in a separated set. The initial bonded-pairs are inserted in the buckets. Such bonded-pairs are then associated to compose new elements that are inserted in the next bucket, according to the corresponding cost, but only if they are allowed functions. These new bonded-pairs are used into the sequence of the associations. The process continues until the target function is found.

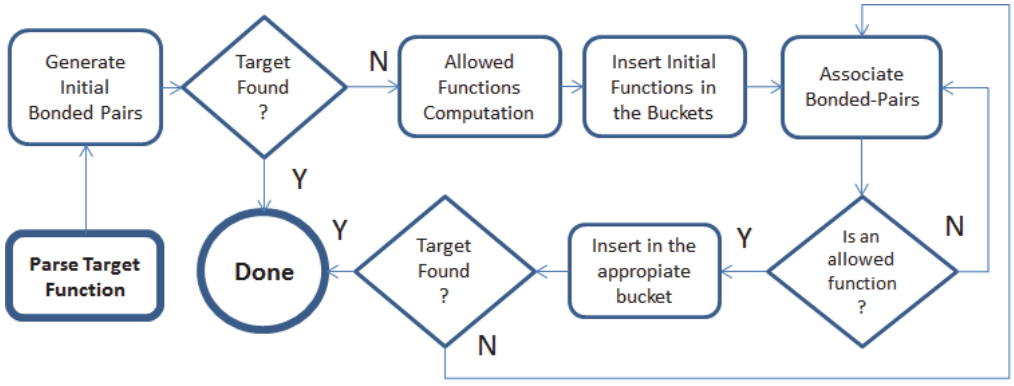


Figure 3.9: General Flow for FC.

3.4 Related Work

The use of a bottom-up approach for logic synthesis was already used by some authors, including (JOSWIAK; BIEGANSKI, 2008) and (HLAVICKA; FISER, 2001).

The FC is different from the work of (JOSWIAK; BIEGANSKI, 2008), because Joswiak does bottom-up synthesis by using information theory (JOSWIAK, 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 (HLAVICKA; FISER, 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 removed from the term anymore, a prime implicant is generated.

Having found a sufficient set of prime implicants, the covering problem is solved. The heuristics are explained in detail in (RUDELL, 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.

3.5 Conclusions

In this Chapter, a novel paradigm for performing logic synthesis, called functional composition (FC) was presented. It is based on a bottom-up approach using composition of Boolean functions to have an efficient cost control. FC has five general principles: bonded-pair representation, initial functions, bonded-pair association, partial order and dynamic programming and allowed functions.

4 MINIMUM DECISION CHAIN COMPUTATION

This chapter discusses the computation of minimum decision chains (MDC). 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) have presented a method to compute MDCs, slightly based on the Quine-McCluskey algorithm (MCCLUSKEY, 1958). This algorithm is referred in this thesis as QMC-MDC (SCHNEIDER ET AL, 2005). (MARQUES ET AL, 2007) used QMC-MDC algorithm as a criterion to evaluate the feasibility of complex gates in a library-free technology mapping approach (REIS; ROBERT; REIS, 1998). However, the execution time of a QMC-MDC algorithm for some kind of functions has been the main drawback. Thus, the computation using FC named here as FC-MDC is proposed to speed-up the MDC computation.

Initially, the MDC concept is discussed. In the following sections, the algorithms QMC-MDC and FC-MDC are presented. QMC-MDC is a top-bottom approach while FC-MDC is a bottom-up approach. Both algorithms are compared in runtime in the section of experimental results.

4.1 Introduction

Logic synthesis methods may rely on the exploitation of Boolean functions properties like positive and negative unateness, binateness and symmetry between variables (WANG; CHANG; CHENG, 2009). One example is the unate recursive paradigm used in the ESPRESSO tool (BRAYTON ET AL, 1984) to decompose functions recursively, leading to easy-to-solve operations on unate sub-functions. Other examples are methods to compute read-once formulas efficiently (GOLUMBIC; MINTZ; ROTICS, 2001; GOLUMBIC; MINTZ; ROTICS, 2008; LEE; WANG, 2007). Indeed, logic synthesis methods use Boolean function properties to guide the algorithms. As a consequence, fast methods to compute Boolean function properties are needed to allow the use of these properties in logic synthesis flows.

There are sum of products minimizers like ESPRESSO-SIGNATURE (MCGEER, 1993) that use the concepts of non-redundant minimal implicants (introduced by (NGUYEN; PERKOWSKI; GOLDSTEIN, 1987)) to avoid the explicit computation of all prime implicants. According to (MCGEER, 1993), "*the work of Perkowski et al. did not receive due attention, possibly because the only algorithm given was that of enumerating all minterms, generating the primes for each, forming the cube of their intersection, and casting out the cubes that are singly contained in any one other*". This is a clear case where an important Boolean function property had its use avoided in practice due to the lack of an efficient algorithm to compute it.

4.2 Minimum Decision Chains

As discussed in Section 2.5.2.2, an ISOP is a SOP where no literal (or group of literals) can be deleted without changing the function represented. The products in an ISOP are prime implicants. To set the output value of the function to logic '1', it is sufficient to assign the values of the variables present in any of the prime implicants, i.e., that at least one of the prime implicants evaluate to logic '1'.

Definition 1: A set of prime implicants that covers a function *f* can be viewed as a set of variable assignments that decide this function.

Definition 2: The largest number of variables in a single prime implicant among a set of prime implicants that decide (cover) a function *f* is the decision chain (DC) of the set.

Lemma 1: Different sets of prime assignments that decide a function *f* are possible, and each set of prime assignments has its own DC.

Definition 3: The minimum decision chain (MDC) of a function *f* is the smallest DC among all possible DCs of a function. The MDC of a Boolean function expresses the largest (worst case) number of variable assignments (i.e. literals in a prime implicant) necessary to cover a minterm of the function.

For example, consider the function of five input variables given by Equation 4.1, which represents a set of five distinct prime variable assignments (prime implicants) that decides (cover) the function. The DC of this set of assignments is four, since the term $b \cdot c \cdot \overline{d} \cdot e$ is the prime implicant with more literals, containing four literals.

$$f(a,b,c,d,e) = \overline{a} \cdot \overline{b} \cdot \overline{d} + \overline{a} \cdot b \cdot \overline{c} + a \cdot \overline{d} \cdot \overline{e} + a \cdot c \cdot d + b \cdot c \cdot \overline{d} \cdot e \tag{4.1}$$

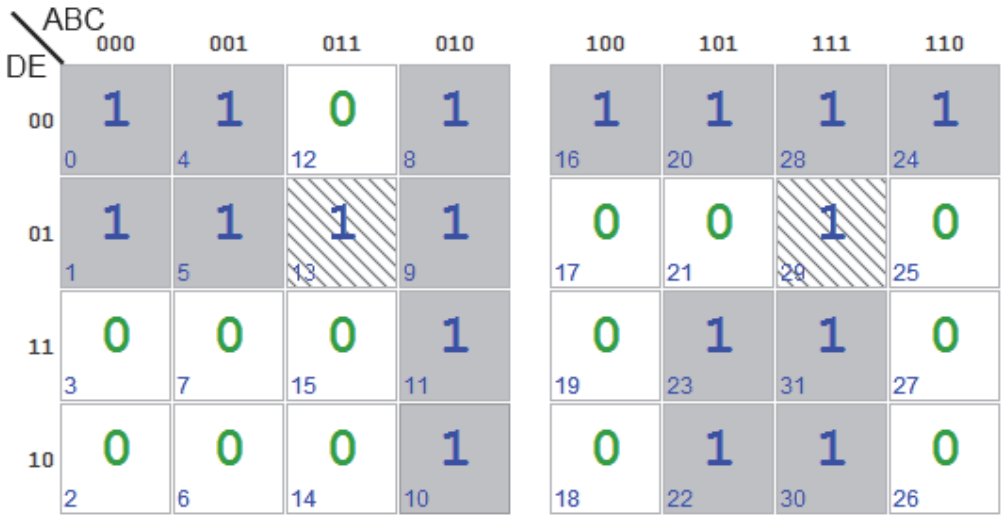


Figure 4.1: K-map for the Eq. 4.1.

The Equation 4.1 can be obtained from the Quine-McCluskey method, and is illustrated as a Karnaugh map in Figure 4.1. The hachured minterms represents the assignment $b \cdot c \cdot \overline{d} \cdot e$. Four variables ($b$=1, c=1, d=0 and e=1) are assigned to cover both minterm 13 ([a,b,c,d,e]=[0,1,1,0,1]) and minterm 29 ([1,1,1,0,1]). However, minterm 13 can be covered by assigning just three variables (a=0, d=0 and e=1) since the minterms

1 ([0,0,0,0,1]), 5 ([0,0,1,0,1]) and 9 (i.e., [0,1,0,0,1]) also belong to the on-set of *f*. In the same way, to cover minterm 29 is necessary to assign only three variables (a=1, b=1 and c=1). As a consequence, Equation 4.1 can be rewritten as Equation 4.2. It is shown as a Karnaugh map in Figure 4.2:

$$f = \overline{a} \cdot \overline{b} \cdot \overline{d} + \overline{a} \cdot b \cdot \overline{c} + a \cdot \overline{d} \cdot \overline{e} + a \cdot c \cdot d + \overline{a} \cdot \overline{d} \cdot e + a \cdot b \cdot c \qquad (4.2)$$

In Figure 4.2, the minterms with horizontal hachure represent the term $\overline{a} \cdot \overline{d} \cdot e$, and the minterms with vertical hachure represent the term $a \cdot b \cdot c$. All selected prime implicants have exactly three literals each. Therefore, the DC of the set of assignments represented by Equation 4.2 is three. Since the largest number of variable assignments (worst case) necessary to cover a minterm of the function *f* is three. Thus, as it cannot be reduced even more for this function, its MDC is three.

The Equation 4.1 was generated using the Quine-McCluskey algorithm. Thus the number of cubes is reduced, compared with the Equation 4.2. The Equation 4.1 has 5 cubes and 16 literals and the Equation 4.2 has 18 literals.

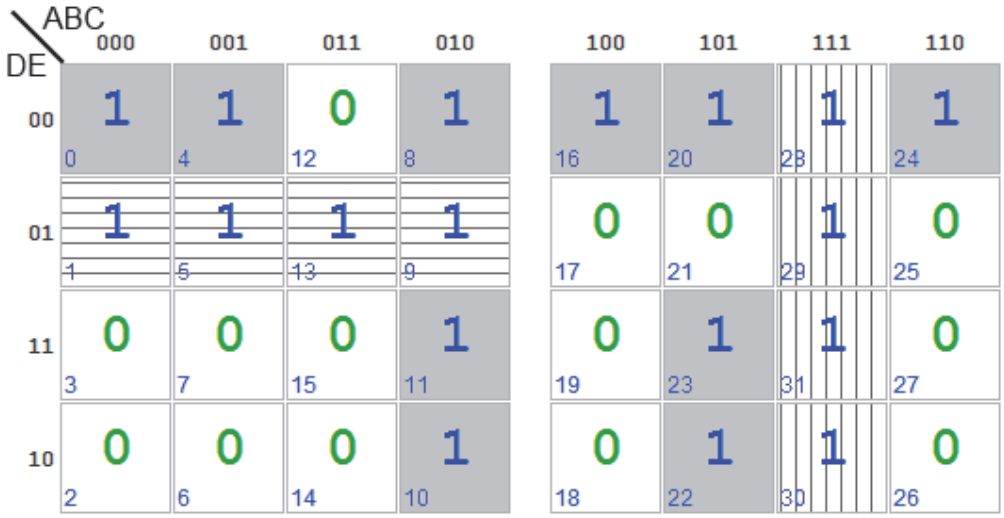


Figure 4.2: Karnaugh map for Equation 4.2.

The MDC of a function *f* is related to the minimum worst case number of series switches required to create a switch network that implements the function *f*. If the MDC is K, then the function requires at least K switches in series to be implemented as a single stage switch network. Also, the existence of a solution with at most *K* series switches is guaranteed. Notice that there is an MDC value for the on-set and another value for the off-set. The MDC value for the on-set is named here as **1-MDC**; while the MDC value for the off-set is named here as **0-MDC**.

It is important to consider if an expression respects the MDC, there is a chance of the amount of literals to increase, even in the factored form. In the Equation 4.1, there are 16 literals, while in Equation 4.2 there are 18 literals. Considering the expressions being represented as switch networks, the switch network represented by Equation 4.2 is potentially faster than Equation 4.1, since the delay is closely related to the largest number of switches series in the stack (WESTE; HARRIS, 2010). Although, the Equation 4.2 has more literals than the Equation 4.1 and this may impact the area.

4.3 QMC-MDC Procedure

The well-known Quine-McCluskey procedure (MCCLUSKEY, 1958) can be adapted to compute the MDC of a function, called in this thesis QMC-MDC (SCHNEIDER ET AL, 2005). It is important to notice that the QMC-MDC method will not return any expression. Indeed, this approach returns a number, which is the MDC of the target function.

The QMC-MDC algorithm makes a modification of the procedure for computing the prime implicants used in the original Quine-McCluskey algorithm. At every step, minterms or cubes are associated to produce larger cubes (larger cubes cover more minterms, therefore have fewer literals). In the original QMC algorithm, the cube generation is done until no prime implicants are found, and no association is possible; in the QMC-MDC algorithm, the prime generation is done until the set of the largest cubes cover the function. In the QMC-MDC procedure starts with a table with only the minterms and the MDC value set to the number of variables of the function. To determine the MDC value, the QMC-MDC considers the cubes produced at every step on the minterms that they cover. At every step, a set of larger cubes is produced. The QMC-MDC verifies if the produced cubes can cover all the minterms of the function by themselves. If this is possible, the MDC value is reduced by one, and the procedure continues. If this is not possible, the execution stops and the number of literals present in the last implicant tested represent the MDC value of this Boolean function. If the function is unate, all cubes are essential prime implicants. Thus the essential prime implicant cube with most literals represents the MDC value of the function.

Using Equation 4.2 as an example, we can see the covering table in Figure 4.3. The lines represent the possible prime implicants, and the columns represent the minterm index. Equation 4.2 can be covered by the cubes containing a rectangle selection, and all cubes have only 3 literals. The algorithm stops the grouping when the target function is covered. The cube with more literals indicates the MDC value.

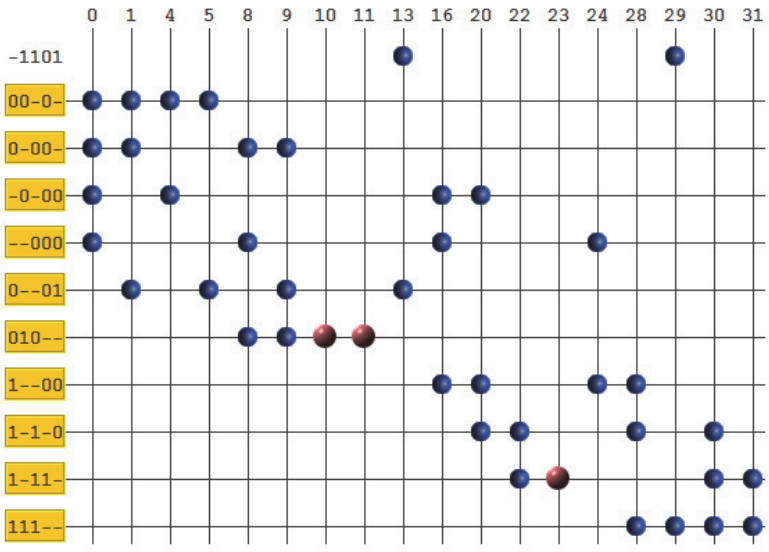


Figure 4.3: Covering table related Equation 4.2.

The QMC-MDC algorithm for a function *f* is efficient when the MDC value is close to the number of variables. However, the QMC-MDC computation can become quite slow for functions with a large number of variables in which the MDC value is much smaller than the number of variables. This behavior is related to many cube expansion for these functions. An alternative approach is proposed based on FC, described further in the next sections.

4.4 FC-MDC Procedure

The functional composition can be used to calculate the MDC of a Boolean function. The algorithm for computing MDC will be referred as FC-MDC. FC-MDC computes the MDC of a Boolean function using a bottom-up approach. Instead of increasing the number of minterms in the implicants by omitting literals from their terms as done in QMC-MDC, the number of minterms in the implicants is gradually decreased by adding new literals in FC-MDC.

The FC-MDC has two initial structures. The first is the accumulator, which is initially the constant zero function. The purpose of the accumulator is to store all smaller functions (in reference to the target function) until the accumulator function is equivalent to the target function. The OR bitwise function is applied to accumulate against the accumulator and the smaller function. The buckets are the second structure, where the larger and not comparable functions are stored.

4.4.1 Functional Composition Setup for MDC Computation

Considering the principles of FC described in Section 3.2, this subsection presents the FC setup (i.e. the choices related to each principle) to compute the MDC of a Boolean function.

1. Bonded-Pair Representation: {Boolean function, MDC value}
2. Initial Functions: Variables complemented and uncomplemented
3. Bonded-Pair Association: Simple association {AND}
4. Partial Order: Number of literals
5. Allowed Functions: Let *f* be a target function, containing *n* variables. The allowed functions are all Boolean space of functions with up to (*n-1*) variables.

4.4.2 FC- MDC Computation

The FC-MDC algorithm starts by checking the relative order of all 1-literal functions (variables complemented and uncomplemented) before inserting them in the 1-literal bucket. If a function is smaller (or equal, in the case the target function has only one variable), this function is accumulated. The remaining functions are combined using the AND bitwise operation, generating 2-literal functions. All smaller functions generated by this combination are accumulated. The generation of 3-literal functions is performed combining 2-literal functions with 1-literal functions. The combination process can be generalized, with the association (using only bitwise AND operation) of (*n-1*)-literal functions with 1-literal functions, generating *n*-literal functions, where *n* is an arbitrary number and smaller than the number of variables This process provides the product terms that compose the function in a SOP form. The upper limit for *n* is the number of variables minus one, since if the algorithm did not stop in the (*n-1*)-bucket,

then the algorithm will certainly stop in the *n*-bucket. Hence, the generation of the *n*-literal bucket can be expressed by Equation 4.3:

$$B_n = \left(B_1 \cdot B_{n-1}\right) \mid n \geq 2 \tag{4.3}$$

4.5 FC-MDC Application Example

In Figure 4.4, there is an execution example of the algorithm considering the function *f* of Equation 4.4:

$$f = a \cdot c + b \cdot c + \bar{a} \cdot b \cdot d \tag{4.4}$$

The first step is to check if the target function is constant. In this case, the algorithm returns zero as the result. The 1-literal bucket is filled with the variables and only the literals with the right polarity are created, reducing the computation time. No accumulation is done at this step because there are no smaller functions. In the 2-literal bucket (with its contents separated using the order criterion), two smaller functions ($a \cdot c$, $b \cdot c$) are found and stored appropriately in the accumulator. In the next iteration, two new smaller functions ($a \cdot c \cdot d$, $\bar{a} \cdot b \cdot d$) are found and also stored in the accumulator. These 4 functions cover all on-set of the function *f* is, and the on-set MDC of this function is three, as the accumulated function became equal to the target function in the 3-literal bucket. Indeed, the (on-set) MDC of a function *f* is the number of literals of the current bucket being processed.

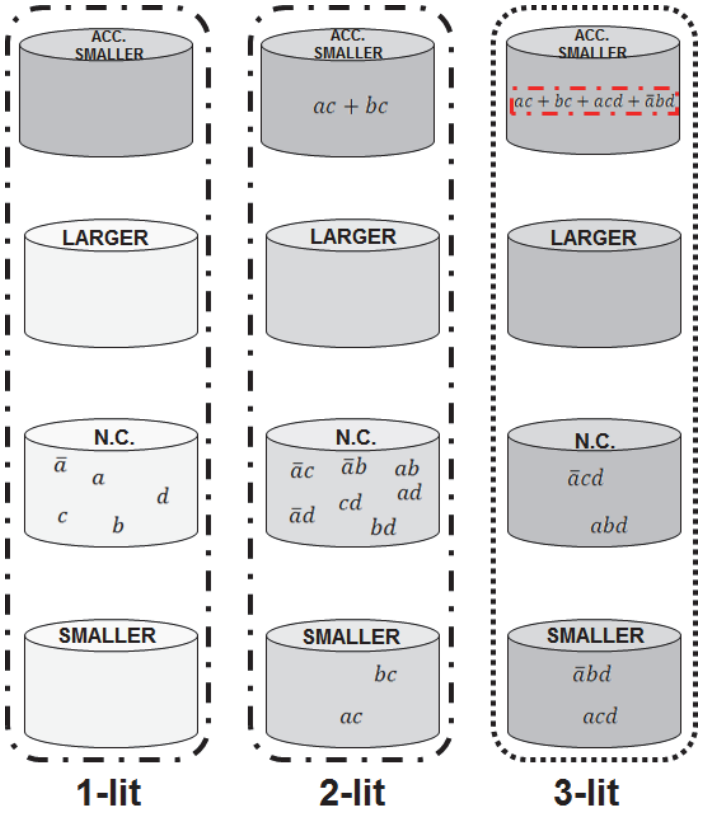


Figure 4.4: MDC computation example.

As an additional feature, the algorithm can be modified to end when the MDC value of the function being computed exceeds a certain threshold limit established by the user. In this case, the exact value of the MDC does not matter, as it exceeded the limit established. The algorithm can stop before finishing all the process if the MDC of the function exceeds the largest value considered feasible by the user, avoiding unnecessary computation of an MDC value not desired. Although if the MDC of the function is greater than the threshold limit, the algorithm returns *"MDC not found"*. Notice that the use of this upper threshold value is only possible due to the way implicants are computed in FC-MDC, i.e. instead of increasing the dimensionality of implicants by omitting literals from their terms, the dimension of a term is gradually decreased by adding new literals. The QMC-MDC algorithm is capable of a lower threshold value, in opposite way of the FC-MDC algorithm, stopping if the MDC of a function is lower than the threshold limit.

4.6 Sum of Products Synthesis Using FC-MDC

The SOP synthesis can be done using the structural part of a bonded-pair as an expression. Storing each bonded-pair in a table can result in products that respect the MDC. To make an ISOP that respects the MDC, each bonded-pair must be stored, and a redundancy check must be done to eliminate redundant products that do not contribute to the final solution. The redundancy check informs if a function is already covered by others. In the SOP found after the FC-MDC execution in Figure 4.4, using a covering table to check redundant primes, $a \cdot c \cdot d$ can be removed, since it is already covered by the terms $a \cdot c$ and $b \cdot c$.

4.7 Experimental Results

Experiments have been carried out to evaluate the efficiency of the proposed FC-MDC method to compute MDCs in comparison to the QMC-MDC approach. The algorithms have been tested in a Core2Duo 2.4 GHz with 4 GB RAM computer.

First of all, there were different sets of functions. The NPN (i.e., input Negation – input Permutation – output Negation) class of representative functions of 4-input and 5-input variables, which contain 222 and 616,125 distinct classes, respectively (CORREIA; REIS, 2001). A third benchmark used in the analysis is the library 44-6.genlib library distributed in the SIS package (SENTOVICH, 1992), which contains 3,503 functions with maximum four series/parallel transistors and with maximum six levels of logic depth in the conventional static CMOS design style. All functions in the 44-6.genlib are read-once.

The QMC-MDC and FC-MDC algorithms were executed in these three set of functions to obtain the on-set MDC values. The execution time is shown in Table 4.1 for both addressed methods and the average on-set MDC value obtained with these sets of functions. The main reason why the FC-MDC algorithm exceeds the QMC-MDC algorithm in performance is associated with the fact that it does not need to fill many buckets when the MDC value is small. Notice that the cases where the MDC is small represent the functions of the more interest in digital design, as they correspond to feasible switch networks (and faster logic gates) with a small number of stacked switches (WESTE; HARRIS, 2010). The QMC-MDC was observed to be slow for computing the MDC of functions in the 44-6.genlib set since this set contains functions

with up to 16 variables, in which it is quite expensive to compute all the prime implicant terms.

Table 4.1: Total execution time of on-set MDC computation.

Function Set	#Functions	QMC-MDC	FC-MDC	Average on-set MDC
4-NPN	222	34 ms	9 ms	3.46
5-NPN	616,125	92 s	104.3 s	4.67
44-6.genlib	3,503	> 4h	5.9 s	3.87

It is very interesting to exploit the property of limitation provided by the FC-MDC, stopping the computation after reaching a certain maximum MDC threshold determined by the user. The execution time improvement due to this feature is illustrated in Table 4.2, which shows the number of functions with an MDC value smaller or equal to a user-defined threshold and the execution time necessary to compute it. FC-MDC considers the pre-defined MDC threshold to stop computation if the MDC of the function is larger than the limit established by the user.

Table 4.2: MDC computation of 5-NPN through FC-MDC method, using a limit value pre-defined by the user.

MDC limit (pre-defined)	Number of functions having MDC = MDC limit	Time to process all 5-NPN
1	1	8.7 s
2	9	21.3 s
3	3,444	44.7 s
4	318,327	90.2 s
5	294,344	104.3 s

In the second phase of experiments, the behavior of QMC-MDC and FC-MDC were evaluated for specific functions, with particular attention to the worst cases obtained with the FC-MDC method that are the best cases of the QMC-MDC method. These cases happen when the functions have minterms that cannot be associated to produce larger cubes. The AND and XOR functions are some related examples. The AND functions have a single minterm, while the XOR functions have no minterm that can be associated into larger cubes. The Karnaugh maps of AND4 and XOR4 are shown in Figure 4.5 and Figure 4.6, respectively.

Experiments were performed to investigate the corner cases of FC-MDC method against QMC-MDC, computing the MDC of AND and XOR functions, from two to eight inputs, considering two situations: (1) full computation, and (2) using the pre-defined threshold limit of MDC equal to four, an industry practical rule of thumb for the maximum transistor stack in digital CMOS integrated circuits (WESTE, HARRIS, 2010).

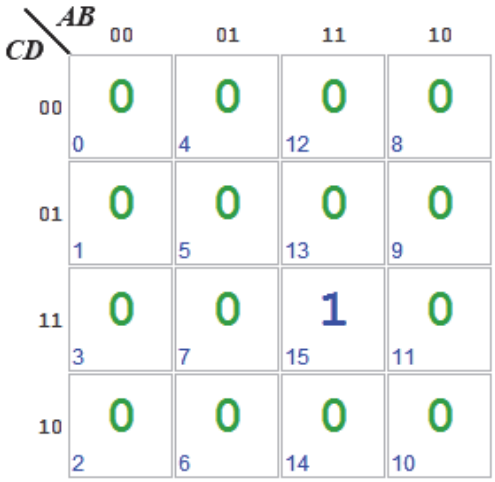


Figure 4.5: Karnaugh map of AND4 function.

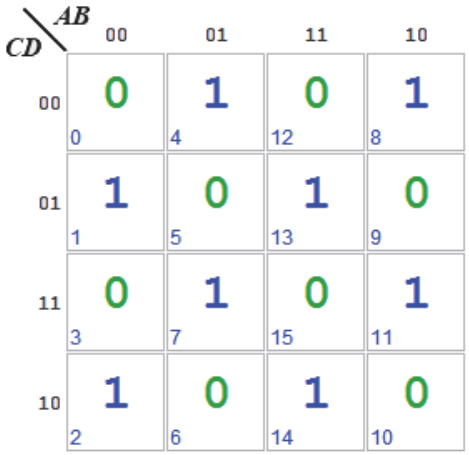


Figure 4.6: Karnaugh map of XOR4 function.

Figure 4.7 and Figure 4.8 provide useful information about the two methods, shown in log scale for better visualization. In Figure 4.7, for the AND function evaluation, the QMC method presents almost constant results, because all functions always have one prime implicant. The FC method, in turn, presents an exponential time increase. In the AND8 case, there is a runtime reduction of *circa* of 50% when exploiting the pre-defined limit parameter (4). In this case, the method is aborted, and a limit overflow result is returned.

The execution time is presented in Figure 4.8 for the XOR function analysis. The QMC-MDC based method presents an exponential runtime increasing since the number of minterms is doubled for each additional variable in the XOR function, by augmenting the number of input variables. On the other hand, the FC-MDC based method presents a more severe increasing than in the AND experiments. However, the pre-defined limit parameter is demonstrated to be very effective, with more than one order of magnitude reduction of total execution time, making the execution time competitive with the QMC-MDC method.

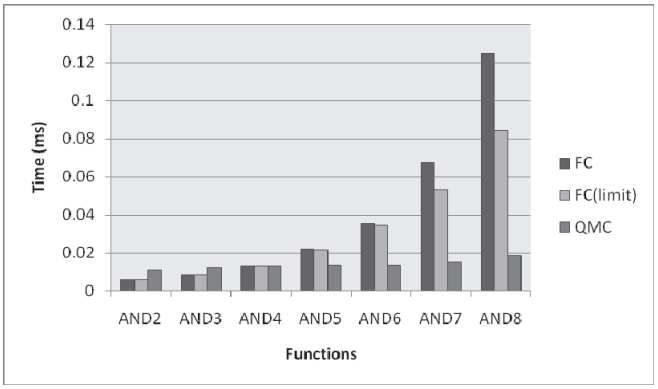


Figure 4.7: MDC computation of AND with 2 to 8 inputs.

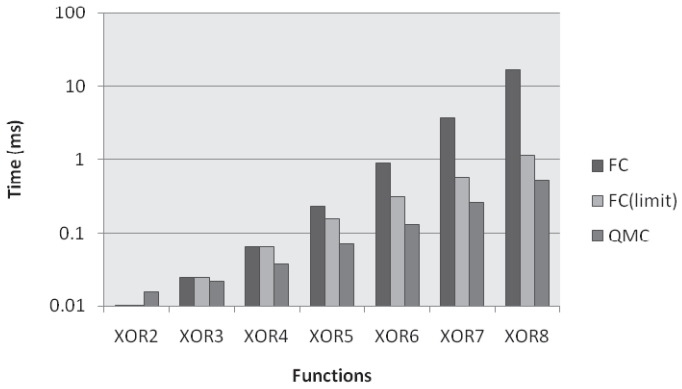


Figure 4.8: MDC computation of XOR, with 2 to 8 inputs.

4.8 Conclusions

This chapter proposed an efficient method to compute minimum decision chains (MDC) of logic functions. The method is referred as FC-MDC. The FC-MDC method is compared to the QMC-MDC method (SCHNEIDER ET AL, 2005), which is a modified version of the Quine-McCluskey algorithm (MCCLUSKEY, 1958). The QMC-MDC method presents some limitations related to the number of prime implicants of the functions, as it has to compute nearly all prime implicants, making it quite computing expensive in some cases. The FC-MDC method is faster in most cases, especially in the cases of CMOS design interest, i.e., logic functions with MDC smaller than 5.

An interesting approach is a use of FC-MDC and QMC-MDC in parallel. When one algorithm finds the MDC value, the other algorithm is aborted. This method will reduce the execution time even more.

5 BOOLEAN FACTORING

The 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 ET AL, 1984) 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.

5.1 Basic Concepts about Factorization

Factoring algorithms can be divided into two groups: those who use algebraic techniques and those who use Boolean techniques.

The algebraic factoring has its basis in the polynomial division. 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 known as a divisor of *f* if *r* is not null, and a factor if *r* is null.

For a given division operation, the resulting *q* and *r* may depend upon the particular representation of *f* and *p*. Moreover, for any logic function, there are many Boolean 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. A restricted version of such division is called algebraic division. For instance, the Equation 5.1 is an algebraic product.

$$f = (x_1 \cdot x_2 + x_3) \cdot (x_4 + x_5) \tag{5.1}$$

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 5.2 is an example of the Boolean product.

$$g = (x_1 \cdot x_2 + x_3) \cdot (\bar{x}_1 + x_2 \cdot x_5) \tag{5.2}$$

Note that Equation 5.1 and Equation 5.2 are different and Equation 5.2 allows products that are not observed in algebraic factoring. If Equation 5.2 is expanded, the Equation 5.3 is found:

$$g = x_1 \cdot x_2 \cdot \bar{x}_1 + x_1 \cdot x_2 \cdot x_2 \cdot x_5 + x_3 \cdot \bar{x}_1 + x_3 \cdot x_2 \cdot x_5 \tag{5.3}$$

The first product can be eliminated by the complementation law ($x \cdot \bar{x} = 0$) and the second product can be simplified by idempotence law ($x \cdot x = x$). The Equation 5.4 is the Equation 5.3, considering the discussed simplifications.

$$g = x_1 \cdot x_2 \cdot x_5 + x_3 \cdot \bar{x}_1 + x_3 \cdot x_2 \cdot x_5 \tag{5.4}$$

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). Usually, the starting expression is an ISOP. Boolean factoring algorithms can start from a functional description or an ISOP. An algorithm that depends on a functional description does not suffer from structural bias.

5.2 Related Work

Since obtaining an optimal (minimal number of literals) factorization for an arbitrary Boolean function is an NP-hard problem, all practical algorithms for factoring are heuristic and provide a correct, logically equivalent formula, but not necessarily a minimal solution. Heuristic techniques have been proposed for algebraic factoring that achieved high commercial success. These include the QuickFactor (QF) and GoodFactor (GF) algorithms available in SIS tool (SENTOVICH et al, 1992).

According to (HACHTEL; SOMENZI, 2006) the only known optimality result for factoring is the one presented by (LAWLER, 1964). Lawler's algorithm starts from a functional description, uses a procedure to compute multilevel prime implicants, but it is too slow, not being able to compute all functions of four variables.

(CARUSO, 1994) proposed an algorithm for Boolean factoring. Its general strategy is similar to the one used by an algebraic factoring algorithm carrying out factoring process by recursively applying the three basic operations called expansion, selection, and reduction but using Boolean identities in the elements. As this algorithm is based on an algebraic method, the results are non-optimal. For example, the Equation 5.5 is taken from (CARUSO, 1994), with 13 literals. The Equation 5.5 can be factored into 10 literals, as presented in Equation 5.6. Equation 5.6 is a literal count improvement over Equation 5.5 of 23%.

$$(a \cdot d \cdot \bar{e} + a \cdot c \cdot \bar{e} \cdot f + \bar{c} \cdot d \cdot e \cdot f) \cdot (b + c) \tag{5.5}$$

$$(a \cdot d \cdot \bar{e} + (d + a) \cdot f \cdot e \cdot \bar{c}) \cdot (b + c) \tag{5.6}$$

Recently, factoring methods that produce exact results for read-once factored forms have been proposed by (GOLUMBIC; MINTZ; ROTICS, 2001). As seen in subsection 2.5.3.1, a function *f* is called read-once if it can be represented by an expression where each variable appears no more than once. They presented an algorithm that uses graph partitioning rather than division. The algorithm is called IROF and is based on algorithms for cograph recognition and on checking normality. The IROF algorithm was extended for any function, called XFactor (MINTZ; GOLUMBIC, 2005). The XFactor algorithm is recursive and operates on the function and on its dual, to obtain the better factored form. The IROF algorithm only works for functions that can be represented by read-once formulas. The Xfactor algorithm is exact for read-once forms

and produces good heuristic solutions for functions that are not included into the read-once class of functions.

A method for exact factoring based on quantified Boolean satisfiability (QBF) was proposed by (YOSHIDA; IKEDA; ASADA, 2006). The Exact_Factor algorithm constructs a special data structure called eXchange Binary (XB) tree, which encodes all equations with a given number N of literals. The XB-tree contains three different classes of configurable nodes: internal (or operator), exchanger and leaf (or literal). All classes of nodes can be configured through configuration variables. The Exact_Factor algorithm derives a QBF formula representing the XB-tree and then compares it to the function to be factored by using a miter structure. If the QBF formula for the miter is satisfiable, the assignment of the configuration variables is computed, and a factored form with N literals is derived. The optimality of the algorithm derives from the fact that it searches for a read-once formula and then the number of literals is increased by one until a satisfiable QBF formula is obtained. This algorithm was extended to incompletely specified logic functions in (YOSHIDA; FUJITA, 2011). The Exact_Factor successfully finds the exact minimum solution for expressions with up 12 literals in 10 minutes, but the algorithm has exponential time in relation to the number of literals to compute the optimal solution (in number of literals).

More aspects may be considered in factoring, besides reducing the number of literals. For instance, logic depth and structural characteristics associated with derived switch networks. Consequently, it is necessary algorithms that can deal with multi-objective design goals, considering topological properties (like the number of series and parallel switches in derived networks) while reducing the number of literals. In this sense, it becomes interesting that factoring algorithms should include:

1. Minimize factored forms taking into account multi-objective goals.
2. Generate more than one alternative solution.
3. Start from a functional description, overcoming the structural bias.

Table 5.1: Comparison between factoring algorithms.

Method	Exactness	Functions treated	Start point	Read-Once optimized	More than one solution	Multi-objective
Lawler	Exact	Up-to 4 variables	Functional Description	No	No	No
Caruso	Heuristic	All	ISOP	No	No	No
Exact_Factor	Exact	Up-to 12 literals	Functional Description	No	No	No
QF/GF	Heuristic	All	ISOP	No	No	No
IROF	Exact	Read-once only	ISOP	Yes	No	No
XFactor	Heuristic	All	ISOP	Yes	No	No
FC-HEURISTIC	Heuristic	All	Functional Description	Yes	Yes	Yes
FC-EXACT	Exact	All	Functional Description	Yes	Yes	Yes

These three important characteristics cited above are present in the factoring algorithms using FC. Table 5.1 shows a comparison of the previous work and the proposed factoring algorithms named herein as FC-HEURISTIC (MARTINS ET AL, 2010) and FC-EXACT (MARTINS ET AL, 2012).

5.3 FC Factoring Baseline Algorithm

The FC factoring baseline algorithm computes new functions from simpler expressions (i.e., with fewer literals) computed in prior steps, to find the target function with fewer literals as possible. 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: $L \leq M$ and $N = L + M$. The procedure combines a L-literal function with a M-literal function, creating a N-literal function by using logic operations. A 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 the Equation 5.7.

$$B_N = \bigcup_{i=1}^{\left\lfloor \frac{N}{2} \right\rfloor} ((B_i * B_{N-i}) \cup (B_i + B_{N-i})) \mid N \geq 2 \tag{5.7}$$

In Equation 5.7, B_k represents the bucket and with index k. 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.

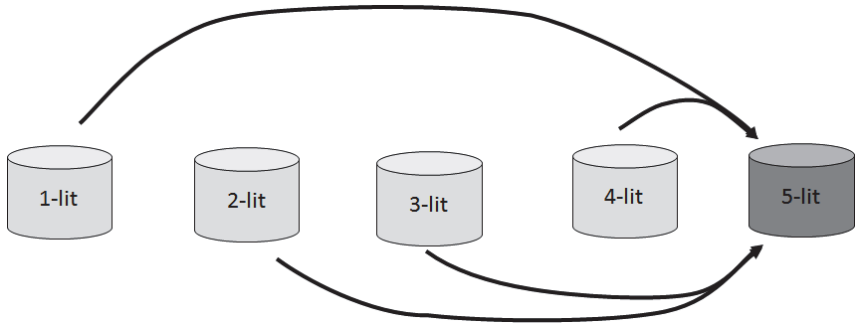


Figure 5.1: Generation of functions contained in the 5-literal bucket

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: Combination of the buckets using Equation 5.7 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 constant functions and functions already present in buckets with smaller indexes are not 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 5.5. 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 2^{2^n} Boolean expressions of *n* inputs. If the functions in each bucket are not pruned, the algorithm becomes unfeasible in memory and computational time. Using the baseline algorithm, both heuristic and exact approaches can be utilized. This chapter will present the heuristic factoring and heuristics to optimize the exact factoring, without losing optimality.

5.4 Functional Composition Setup for Boolean Factoring

Considering the principles of FC described in Section 3.2, this section presents the FC setup for factoring of Boolean functions.

1. Bonded-Pair Representation: {Boolean Function, Expression}
2. Initial Functions: Variables
3. Bonded-Pair Association: Simple association {AND/OR}
4. Partial Order: Number of literals
5. Allowed Functions:
    - Read-Once: Cofactors and cube cofactors.
    - Heuristic: Cofactors, cube cofactors, and combinations of cofactors and cube cofactors.
    - Exact: All functions are accepted

5.5 Boolean Operations Considering Order

To reduce the amount of functions in the buckets, it is necessary to analyze the order of the functions generated compared to the target function. The order is important to define which functions are important in the generation of the target function. The functions classified by their order against the target function are shown in Figure 5.2.

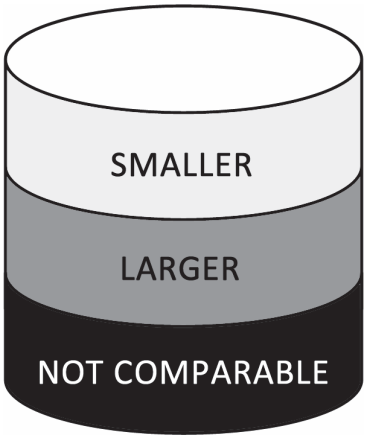


Figure 5.2: Bucket divided in smaller, larger and not comparable sets.

Considering the generated functions separated by their respective order and applying the AND and OR operations between them, 20 unique operations can be done, as seen in Table 5.2. Notice that the number of all possible combinations is 32, 2 by 2. However, since Boolean operations are commutative, the position of the functions with respect to the operator does not matter, resulting in 20 unique operations. In Table 5.3 the results of the operations of Table 5.2 are described.

Table 5.2: AND/OR operations considering function order.

#	AND	#	OR
(1)	$SM \cdot SM$	(11)	$SM + SM$
(2)	$SM \cdot LG$	(12)	$SM + LG$
(3)	$SM \cdot NC$	(13)	$SM + NC$
(4)	$SM \cdot DJ$	(14)	$SM + DJ$
(5)	$LG \cdot LG$	(15)	$LG + LG$
(6)	$LG \cdot NC$	(16)	$LG + NC$
(7)	$LG \cdot DJ$	(17)	$LG + DJ$
(8)	$NC \cdot NC$	(18)	$NC + NC$
(9)	$NC \cdot DJ$	(19)	$NC + DJ$
(10)	$DJ \cdot DJ$	(20)	$DJ + DJ$

Table 5.3: Description of Table 5.2 operations results.

Operation Number	Operation Result Description
(1),(3)	A smaller function with fewer minterms than the composing functions or the constant zero function.
(2)	The smaller composing function.
(4)	The constant zero function
(5)	A larger function with fewer minterms than the composing functions

	or the compared function.
(6)	A smaller function or a not comparable function.
(7),(9),(10)	The constant zero function or a disjoint function.
(8)	A smaller function or a not comparable function or a disjoint function or the constant zero function.
(11)	An SM function with more minterms than the composing functions or the compared function.
(12)	The composing LG function.
(13),(18)	A larger function or a not comparable function.
(14)	Generates a not comparable function.
(15),(16)	A larger function with more minterms than the composing functions.
(17)	The larger composing function or larger function with more minterms than the composing functions.
(19)	A not comparable function with more terms in the off-set of the target function or the composing a not comparable function.
(20)	A disjoint function with more terms in the off-set of the compared function.

5.6 Read-Once Factoring

The read-once factoring algorithm is one of the derived versions of the baseline algorithm. It relies on the symmetry of the variables. It groups variables that are symmetric or anti-symmetric testing using bitwise AND/OR operations and inserting in the respective bucket.

The allowed functions are only the cofactors and cube cofactors. There are at most v buckets, where v is the number of variables of the target function. If the target function is not found in the last bucket, then the target function is not a read-once function.

Theorem: If a function f can be represented by a read-once formula, all the partial sub-equations in the formula correspond to functions that are cube cofactors of f.

Proof: *As each variable appears as a single literal, they can all be independently set to non-controlling values, which makes only one literal disappear at a time. Any sub-equation (or sub-set) of f can be obtained by assigning non-controlling values to the variables to be eliminated. These variable assignment forms are cube cofactors by definition.*

5.7 Exact Approach

The exact approach gives minimum literal count factored forms. In this approach, all functions need to be allowed. The number of functions grows exponentially in each bucket and optimizations are needed to safely eliminate functions that do not contribute to the minimal solution.

5.7.1 Allowed Combinations

For the FC-EXACT algorithm, analyzing the 20 possible AND/OR operations are shown in Table 5.3, the two ways to reach the target function are (5) and (11) operations. This is expected, since in (5), the AND operation reduces the minterms from two larger functions and, in (11), the OR operation increases the minterms from two smaller functions.

Disjoint functions do not help in combination process since they do not have shared terms with the solution function. In this sense, the operations that lead to disjoint functions (4, 7, 9, 10, 14, 17, 19, 20) can be avoided and safely discarded.

The combinations (1, 2, 3) can also be discarded, since there is a minterm reduction, tending to the constant zero function and deviating from the solution. Similarly, (12, 15, 16) combinations generate larger functions with minterm gain, tending to the constant one function and deviating from the final solution. From all 20 operations, only (5, 6, 8, 11, 13, 18) are necessary for the combination process to reach the solution.

5.8 Heuristic Approach

The FC-EXACT approach sometimes is not time feasible, so the heuristic approach was developed to overcome this issue. 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.

5.8.1 Allowed Functions

In the FC-HEURISTIC algorithm, only the smaller, larger and not comparable functions are considered, as the disjoint functions are discarded, because they do not contribute to the solution, as discussed in Section 5.7.1.

The allowed functions in FC-HEURISTIC are a set of functions derived from the cofactors[1] of the target function. The initial step is extracting all cofactors of the target function to associate them in the next step. The association of cofactors is illustrated in Figure 5.3. Not comparable cofactors are associated using AND/OR bitwise 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 bitwise operation, only storing resulting larger functions. Similarly, there is the association among not comparable and larger cofactors using AND bitwise 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 obtain good sub-expressions of the formula, by setting variables to zero and one in an optimized factored form.

[1] For simplification sake the term cofactor represents the cofactors and cube cofactors of a function in this section, unless otherwise noted.

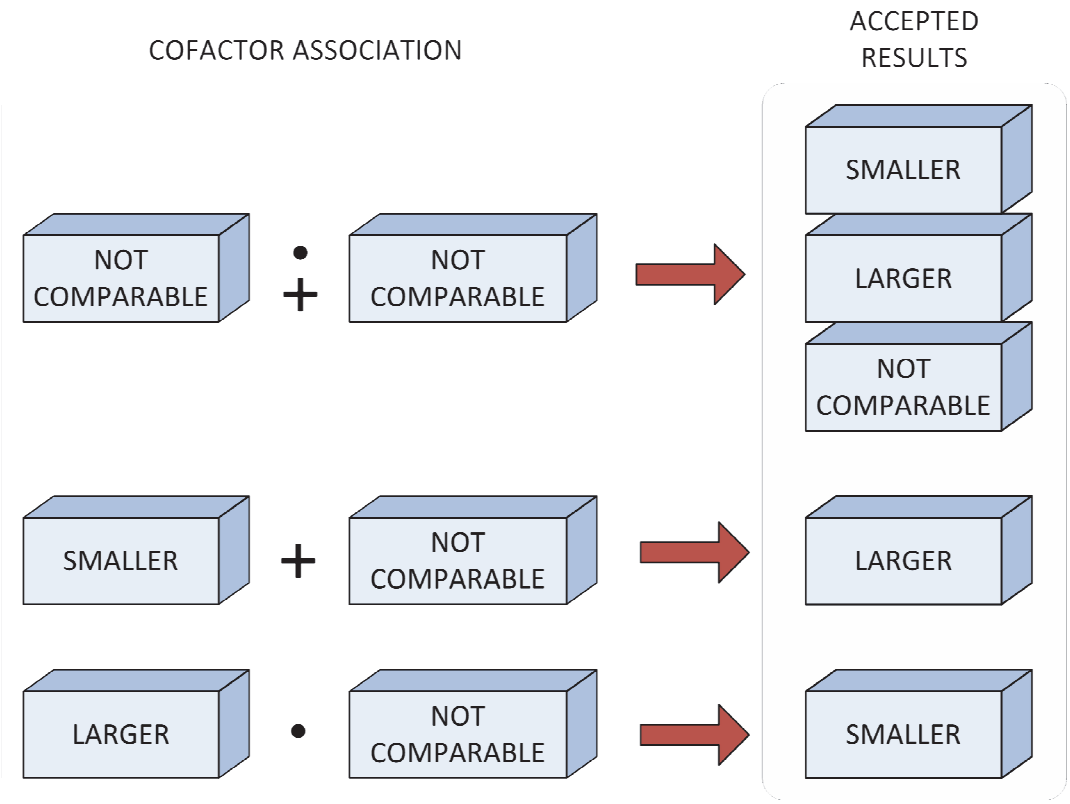


Figure 5.3: Allowed Functions generation.

5.9 Other Optimizations

There are other important optimizations. In the factoring execution, only the initial functions have the implementation cost known. Depending on the factoring objective, as the structural parts of the functions are discovered, some functions in the search space may not attend the constraints desired. In this case, they cannot be inserted in the buckets. An "already abandoned" set can store these functions.

For instance, an abandon criteria can be the MDC, presented in Chapter 4. Functions that do not respect the MDC are discarded, reducing the number of combinations and reducing the execution time.

5.10 General Flow

Figure 5.4 shows the flow chart for the FC-HEURISTIC algorithm. The first step is to check if the target function is constant. In this case, the algorithm returns the constant. The polarity of variables is calculated by checking the unateness. If there are no binate variables, the read-once algorithm is called. If the function is read-once, the optimal result is obtained. If the function is not read-once, the algorithm starts to associate functions and checking if the resulting function is allowed, allowing it to insert this function in a next bucket.

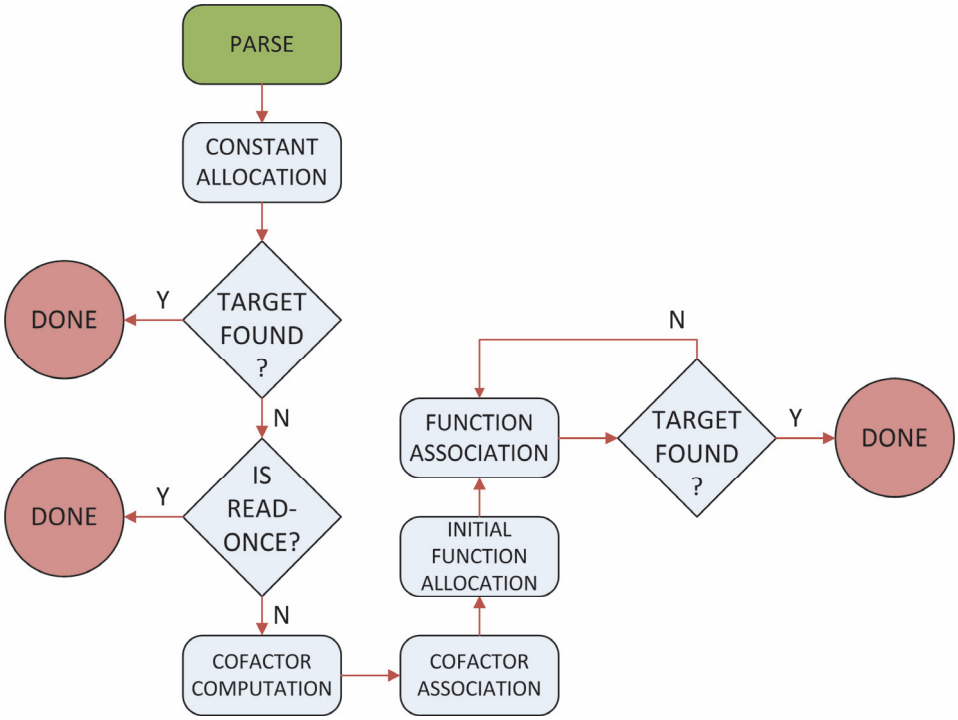


Figure 5.4: Heuristic approach flow.

In the "cofactor association" step, all cofactors and cube cofactors are generated and separated in sets by its relative order against the target function (not comparable, smaller and larger). The "cofactor combination" generates all allowed functions for the FC-HEURISTIC algorithm. In FC-EXACT, this step and "cofactor combination" are ignored, since all functions are allowed.

After the initialization of the allowed functions, the algorithm proceeds with the initial functions allocation. It starts by creating the 1-literal functions. For the 1-literals functions, only the literals with the right polarity are created, reducing the computation time. If the target function is not found in the 1-literal bucket, subsequent buckets following the bucket generation schema are generated, until the target function is found. A solution is always an OR (AND) operation between smaller (larger) subfunctions in previous buckets and the functions in the current smaller (larger) bucket. The algorithm does not need to stop in the first solution; multiple solutions with different costs can be found.

5.11 Examples

Two examples are given in this section. The first demonstrates the read-once algorithm and the second demonstrates the FC-HEURISTIC algorithm.

5.11.1 Read-Once Algorithm

A function *r is* given as an example to illustrate the read-once algorithm. This function is a simple example in SOP form, seen in Equation 5.8.

$$r = a \cdot c + b \cdot c + d \tag{5.8}$$

After preprocessing (parse, constant checking), the variable polarities and symmetry information are computed. All variables are positive unate, and there is a symmetry group $\{a,b\}$. This information is used to reduce the number of cofactors and cube cofactors. The computation of the cube cofactors results in different functions listed in Table 5.4. Some cofactors and cube cofactors are excluded by the symmetry group information. For instance, $r_{a=0} = b \cdot c + d$ is excluded because have only the variable 'b'.

Table 5.4: Cofactors and cube cofactors of *r*, simplified by symmetry information.

Cofactors	Cube Cofactors
$r_{a=1} = c + d$	$r_{a=1,d=0} = c$
$r_{c=1} = a + b + d$	$r_{c=1,d=0} = a + b$
$r_{c=0} = d$	----
$r_{d=0} = (a + b) \cdot c$	----

The '*c*' and '*d*' functions are inserted in the 1-literal bucket, and $a + b$ function is inserted in the 2-literal bucket.

The combination step is illustrated in Figure 5.6. The 1-literal functions are combined producing the 2-literal combinations. Only subfunctions that are in the allowed subfunctions hash are accepted as intermediate subfunctions. The combination continues until the 4-literal bucket, where a solution is found. The $(a + b) \cdot c$ function located in the 3-literal bucket is associated using OR operation with d function, in the 1-literal bucket, finding the target function.

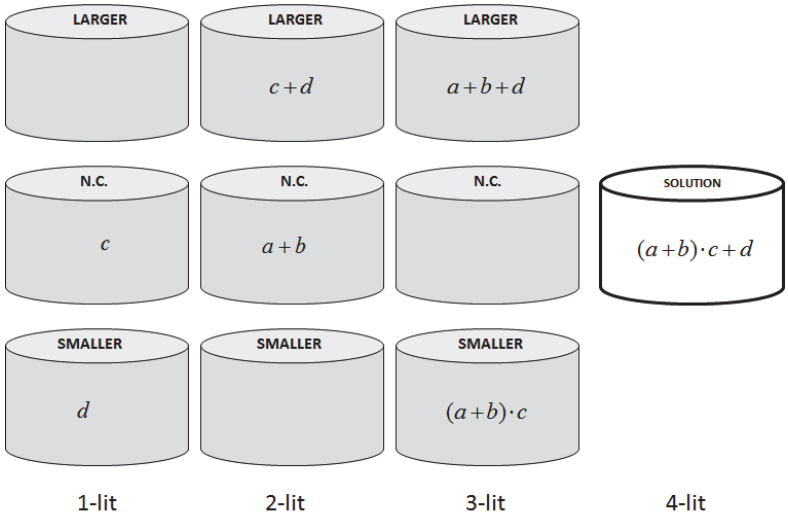


Figure 5.5: Combination step of factorization with simplified bucket content.

5.11.2 FC-HEURISTIC Algorithm

A function *f* in (5.9) *is* given as an example to illustrate the FC-HEURISTIC. This function is a simple but illustrative example in SOP form, seen in Equation 5.9.

$$f = a \cdot c + b \cdot c + \bar{a} \cdot b \cdot d \tag{5.9}$$

After preprocessing (parse, constant checking), the function is checked if it is read-once. This step fails because the function has a binate variable ('a'). The algorithm computes the initial functions. This step computes the unateness and symmetry information for all variables. Variable 'a' is binate and variables 'b', 'c' and 'd' are positive unate in respect to the function *f*. No variable is symmetric, so symmetry information is not used to reduce the computation of cube cofactors. All 1-literal functions are inserted in the 1-literal bucket. The computation of the cube cofactors results in different functions listed in Table 5.5.

Table 5.5: Cofactors and cube cofactors of *f*.

Cofactors	**Cube Cofactors**
$f_{a=1} = c$	$f_{a=0,b=1} = c + d$
$f_{a=0} = b \cdot (c + d)$	$f_{a=0,c=1} = b$
$f_{b=1} = c + \bar{a} \cdot d$	$f_{a=0,c=0} = b \cdot d$
$f_{b=0} = a \cdot c$	$f_{a=0,d=0} = b \cdot c$
$f_{c=1} = a + b$	$f_{b=1,c=0} = \bar{a} \cdot d$
$f_{c=0} = \bar{a} \cdot b \cdot d$	$f_{b=0,c=1} = a$
$f_{d=1} = (a + b) \cdot (c + \bar{a})$	$f_{c=0,d=1} = \bar{a} \cdot b$
$f_{d=0} = c \cdot (a + b)$	$f_{a=0,b=1,c=0} = d$
----	$f_{b=1,c=1,d=1} = \bar{a}$
----	$f_{b=1,d=1} = c + \bar{a}$

It is important to make two observations: the total number of cube cofactors is greatly reduced since some cube cofactors represent the same logic function, and also some of them are constant functions. The second observation is all cofactors and cube cofactors contain the literals in the same polarities of the function *f*.

The combination step is illustrated in Figure 5.6, which is a simplified version of the process for better legibility (each bucket has much more functions allocated). The 1-literal functions are combined producing the 2-literal combinations. Only subfunctions that are in the allowed subfunctions hash are accepted as intermediate subfunctions. The combination continues until the 5-literal bucket, where a solution is found. The $a + b$ function located in the 2-literal bucket is associated using AND operation with $c + \bar{a} \cdot d$ function, in the 3-literal bucket, finding the target function.

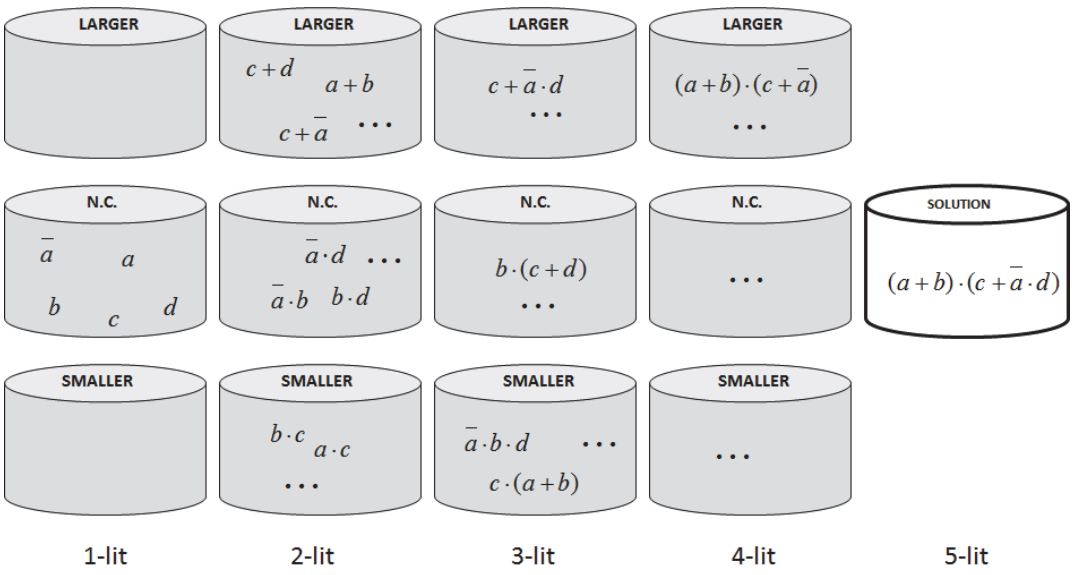


Figure 5.6: Combination step of factorization with simplified bucket content.

The implementation can also have associated some data about number of literals, number of transistors, series/parallel properties, etc. These data are necessary to factorize a target function considering multi-objective goals (i.e. minimize the number of transistors involved while respecting the MDC function).

5.12 Experimental Results

The experiments are divided into three parts. The first experiment focuses in the multi-objective factorization. The second experiment shows the factorization quality over a set of all 4-input functions grouped in 3,984 P-class functions. The last experiment shows the factorization quality in some functions of ISCAS'85 benchmarks. The experiments have been carried at a computer with a Core2Duo 2.4 GHz processor and 4 GB RAM.

The multi-objective factorization allows controlling the costs. Table 5.6 presents an example of a function of 6 input variables that was factored using the multi-objective factorization. The '**L**', '**S**' and '**P**' are literals, series switches and parallel switches, respectively. Equation 5.10 is the minimum literal count logic function obtained when only literals are minimized. Equation 5.11 is obtained when the algorithm is required to minimize literals and respect the on-set MDC (which is 3). Equation 5.12 is obtained when the algorithm is required to minimize literals and respect the off-set MDC (which is 4). In this case, the number of literals is increased by two, from 20 to 22 literals.

Notice that, as the data of subfunctions is always known during the execution of the algorithm, the algorithm proposed can be modified to accept only subfunctions with certain characteristics. The approach can consider any secondary criteria that can be computed in a monotonically increasing way so that the solutions are generated in the right order. Additionally, the new costs must be easily obtainable for a combination of the subfunctions. In the case of literals, this can be done by simple addition. These requirements allow not only controlling the number of series and parallel switches, but also logic depth (per input variable) and function support size.

Table 5.6: Results of multi-objective goal factorization.

Eq.	Logic Function	L	S	P	Time
(5.10)	$f \cdot e + ((d+c)(b+a)(d \cdot c + b \cdot a) + ((f+e)(d \cdot c + b \cdot a + ((d+c)(b+a)))))$	20	4	7	1.15s
(5.11)	$f \cdot e + ((d \cdot c \cdot (b+a)) + (b \cdot a + (d+c)) + ((f+e)(d \cdot c + b \cdot a + ((d+c)(b+a)))))$	20	3	9	1.34s
(5.12)	$(((e+f)(e \cdot f + c \cdot d)) + a + b)((e + f + a + (c \cdot d \cdot ((((f+e)(e \cdot f + a \cdot b)) + c + d)))))$	22	8	4	1.22s

The second experiment was performed on the set of all 4-input functions (grouped in 3,984 P-class functions (CORREIA; REIS, 2001), by equivalence through input permutation). The number of literals after mapping with a commercial library was investigated. In a second experiment, area optimization was investigated (REIS, 1999; CORREIA; REIS, 2004; TOGNI ET AL, 2002). They are demonstrated in Table 5.7.

Table 5.7: Results regarding number of literals and area after mapping.

Method	**Literals**	**Literal Difference over FC-EXACT**	**Area Difference over FC-EXACT**	**Execution time**
FC-EXACT	36028	-	-	16h
FC-HEURISTIC	36738	+1.97%	+3.77%	114s
QF (SENTOVICH ET AL, 1992)	38341	+6.42%	+6.61%	47s
GF (SENTOVICH ET AL, 1992)	37893	+5.18%	+4.67%	47s
ABC (BERKELEY, 2012)	38246	+6.16%	+7.4%	2s
XF (MINTZ; GOLUMBIC, 2005)	37652	+4.51%	+1.1%	24s

Five different methods from the literature are compared to the two methods proposed in this thesis. The five methods are referred as QF (QuickFactor) (SENTOVICH ET AL, 1992), GF (GoodFactor) (SENTOVICH et al., 1992), ABC (BERKELEY, 2012), XF (X-Factor) (MINTZ; GOLUMBIC, 2005). The QF and GF are from the SIS package. The results from QF, GF and ABC, were generated using the "*print_factor*" command. The results from the row X-Factor in Table 5.7 are obtained by an in-house implementation of the algorithm presented in (MINTZ; GOLUMBIC, 2005). Since these methods are heuristic, they present some overhead regarding number of literals when compared to FC-EXACT. This overhead increases the final area. However, there are other criteria in technology mapping to consider besides literals, as the variable order and the expression balancing (HASSOUN; SASAO, 2002; MISHCHENKO ET AL, 2011)

In the last experiment, a comparison with some MCNC (YANG, 1991), an analysis with benchmark functions is performed. Table 5.8 shows the number of literals for some benchmark functions where our algorithm produces better or equal literal count than QuickFactor, GoodFactor, ABC, and X-Factor. The column SOP represents a Quine-McCluskey minimization (no factorization) and the column FC represents the FC-EXACT and FC-HEURISTIC results.

Table 5.8: Number of literals after factorization in some benchmarks.

Logic Function	SOP	QF	GF	ABC	XF	FC
b9_a1	56	12	12	12	12	**12**
rd53_0	20	14	14	14	12	**12**
rd53_1	80	28	28	28	46	**28**
cm162a_o	29	16	16	16	13	**12**
cm162a_p	36	18	18	16	14	**14**
cm162a_q	43	20	20	18	16	**16**
cm163a_r	31	13	13	13	13	**12**

5.13 Conclusions

This chapter proposed two Boolean factoring algorithms. One of the algorithms is the first multi-objective factoring algorithm. The algorithm can take secondary criteria (like series and parallel number of switches, or support size) into account, while generating several alternative solutions. This characteristic makes it a useful piece for approaches based on restructuring small portions of logic, like (WERBER; RAUTENBACH; SZEGEDY, 2006) and (MISHCHENCKO; BRAYTON; CHATTERJEE, 2008). The unique characteristics of the algorithm make it very useful in the context of local optimizations. From an execution time point of view, the algorithm is slower compared to other approaches, but still feasible. From a quality point of view, the proposed algorithm always delivered superior (or equal) results compared to other approaches.

The second algorithm is an exact algorithm for factoring Boolean functions, delivering optimal results in number of literals. This algorithm is slower than the other algorithms presented in the literature. A possible and practical application is the generation of a look-up table to optimize 4-input subcircuits. This algorithm also can be multi-objective, allowing optimal minimization in multiple criteria.

6 BOOLEAN FACTORING USING XOR

Traditional logic optimization methodology based on (BRAYTON ET AL, 1984; SENTOVICH ET AL, 1992) has emerged as a dominant method for logic synthesis. This optimization methodology generates good results for AND/OR functions of control and random logic. However, results are not satisfactory for arithmetic and logic functions that could benefit from the inclusion of exclusive-OR operation.

Boolean functions using XOR operator received less attention in the logic synthesis research. The "*technology independent optimization*" step in logic synthesis aims to reduce the number of literals in logical expressions. The number of literals is a good metric related to the number of logic gates used in the technology mapping process.

Technology mapping transforms a technology independent logic network into a netlist of technology dependent logic gates (HASSOUN; SASAO, 2002). Technology mapping process relies on static pre-characterized libraries. Each cell of the library is fully characterized through exhaustive simulations, resulting in accurate information about timing, power consumption, and physical area. Hence, such technology mapping algorithms are restricted to use these cells available in the target library.

As the quality of the mapping will depend significantly on the expression representing the circuit, the best algorithm executed over a poorly factored expression may produce a worse result than an average quality mapping over a minimal factored expression. This problem is known as structural biasing (CHATTERJEE ET AL, 2005). In this sense, the benefit of factorization taking into account XOR operator is to generate more reduced number of literals that will generate smaller circuits (even considering a higher physical implementation cost of the XOR gate)

6.1 Related Work

Perkoswi (SONG; PERKOSWI, 1998) proposed an algorithm that factorizes an expression using AND/OR/NOT/NOR/NAND/XOR/XNOR operators. This algorithm has an exclusive-SOP (ESOP) as input. However, there is no efficient way to find a minimal ESOP (SASAO, 1999). Moreover, Perkoswi considers the XOR operation has the same cost of implementation of an AND/OR operation, probably representing an area impact in standard cell designs.

Sasao (SASAO, 2005) implemented an algorithm that generates a 3-level expression result (two SOPs joined by a XOR). The AND-OR-XOR three-level network is suitable for implementing arithmetic functions, being one of the simplest gate arrangements since it contains only a single two-input XOR gate. For instance, the use of two-input XOR gates at the outputs of PLA efficiently realize adders (WEINBERGER, 1979).

An algorithm to generate multilevel expressions containing AND/OR/NOT/XOR was proposed by Sasao (SASAO ET AL, 1995). It takes advantage from pseudo-Kronecker decision diagram (PKDD) reduced structure to represent a logic function. The PKDD is a modified BDD that allows Shannon, positive and negative expansions for each variable. In Figure 6.1, an example of a pseudo-Kronecker tree, where the first variable uses the Shannon expansion. The second variable uses both the positive and negative Davio expansions, and the last variable uses all the three expansions, where 'S' represents the Shannon expansion, 'pD' the positive Davio expansion and 'nD' the negative Davio expansion.

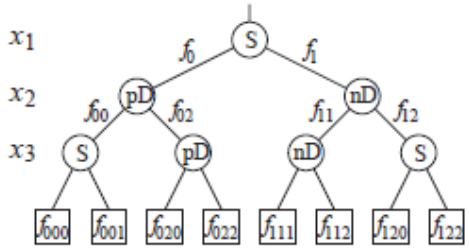


Figure 6.1: PKDD representation for a Boolean function.

Turton (TURTON, 1996) and Tinder (TINDER, 1995) proposed XOR based synthesis methods focused on educational applications on top of Karnaugh maps. Turton proposed a new representation for minterms, extending the Quine-McCluskey algorithm to generate expressions using XOR operation. Tinder proposed patterns for XOR recognition in a Karnaugh Map. Both applications are focused for educational use, and they do not worry about the quality of results and execution time.

The proposed method to factorize expressions using XOR operator (named herein FC-EXACTXOR) is an extension of the FC-EXACT algorithm. FC-EXACTXOR inherits all FC-EXACT features, like multi-objective goals, more than one solution and functional description as the start point. The multi-objective characteristic allows FC-EXACTXOR to provide costs to operators. Since XOR operator has a higher physical implementation cost, FC-EXACTXOR can avoid or reduce the use of XOR operator, increasing its importance in digital designs. Table 6.1 shows a comparison between the previous works and the proposed factoring algorithm using XOR.

6.2 Functional Composition Setup for Boolean Factoring considering XOR

Considering the principles of FC, described in Section 3.2, this section presents the FC setup for factoring of Boolean functions including the XOR operator, being similar to the FC setup for FC-EXACT:

1. Bonded-Pair Representation: {Boolean Function, Expression}
2. Initial Functions: Variables
3. Bonded-Pair Association: Simple association {AND/OR/XOR}
4. Partial Order: Number of literals
5. Allowed Functions: All functions are accepted

Table 6.1: Comparison between factorization using XOR algorithms.

Method	Exactness	Start point	Levels	Has XOR different weight?	Multi Objective	More than 1 solution
(SONG; PERKOSWI, 1998)	Heuristic	ESOP	Multi-level	No	No	No
(SASAO, 2005)	Heuristic	Functional Description	3	No	No	No
(SASAO ET AL, 1995).	Heuristic	PKDD	Multi-level	Yes	No	No
(TURTON, 1996)	Heuristic	Functional Description	2	No	No	No
(TINDLER, 1995)	Heuristic	Functional Description	2	No	No	No
FC-EXACT-XOR	Exact	Functional Description	Multi-level	Yes	Yes	Yes

6.3 Exact Factoring with XOR

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

$$B_n = \bigcup_{i=1}^{\left\lfloor \frac{N}{2} \right\rfloor} \left((B_i * B_{N-i}) \cup (B_i + B_{N-i}) \cup (B_i \oplus B_{N-i}) \right) \mid N \geq 2 \tag{6.1}$$

At first sight, this can be seen as an increase in execution time, since there is one more operation to be done against each pair of functions However the XOR operator reduces the number of intermediate subfunctions. For instance, the importance factorization considering XOR operation is demonstrated considering the XOR4 function as example. The minimal factored form (in number of literals) is represented as follows:

$$u = \left(\left((b + d) \cdot (\bar{b} + \bar{d}) \right) + \left((a + c) \cdot (\bar{a} + \bar{c}) \right) \right) \cdot \left(\left((b + \bar{d}) \cdot (\bar{b} + d) \right) + \left((a + \bar{c}) \cdot (\bar{a} + c) \right) \right) \tag{6.2}$$

Using the FC-EXACTXOR algorithm, with the XOR operation the minimal factored form is represented in Equation 6.3.

$$v = a \oplus b \oplus c \oplus d \tag{6.3}$$

6.3.1 Boolean Operations Considering Order

Table 5.2 can be extended to the XOR bitwise operation. There are now 10 new unique operations that can be done, as seen in Table 6.2. These operations are described in Table 6.3

Table 6.2: XOR operation considering function order.

#	XOR
(21)	$SM \oplus SM$

(22)	$SM \oplus LG$
(23)	$SM \oplus NC$
(24)	$SM \oplus DJ$
(25)	$LG \oplus LG$
(26)	$LG \oplus NC$
(27)	$LG \oplus DJ$
(28)	$NC \oplus NC$
(29)	$NC \oplus DJ$
(30)	$DJ \oplus DJ$

Table 6.3: Result of the operations in Table 6.2.

# Operation	Operation Result
(23)	An LG function or a not comparable function.
(22),(24) (26)	Generates a not comparable function.
(21)	If there are minterms in common in the two composing functions, the result function is an SM function with fewer minterms. Otherwise, the operation gives the same result of (11)*.
(25), (30)	A disjoint function.
(27)	If there are common minterms in the off-set of the target function, it can generate the target function or an LG function. Otherwise, operation gives the same result of (17)*.
(28)	Any function, including the target function.
(29)	If there are common minterms, the combination can generate an SM function or a not comparable function. Otherwise, the operations give the same result of (19)*.

*- these references are located in Table 5.3.

6.3.2 Allowed Combinations

As the FC-EXACTXOR algorithm uses AND/OR operations, all the operations described for FC-EXACT needs to be considered. Analyzing the 10 XOR combinations in Table 6.3, we can observe that only (27) and (28) can find the target function, besides the combinations used for FC-EXACT. In this case, all functions that generate disjoint results in FC-EXACT needs to be considered and added. The combinations (21), (22) and (23) deviate from the target function or generates the same results of the equivalent OR operation (11), (12) and (13), so the combinations with XOR can be discarded. From all 10 operations of Table 6.2, the combinations (24), (25), (26), (27), (28), (29) and (30) are needed in the combination process to find the solution for the FC-EXACTXOR algorithm.

6.3.3 Condition to enable XOR factorization

It is verified empirically that the XOR operation is only useful when there are at least 2 binate variables. In this case, the FC-EXACTXOR can be efficiently exploited if there are at least 2 binate variables, the FC-EXACTXOR can be used. 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 (i.e.: $a \oplus b = \bar{a} \cdot b + a \cdot \bar{b}$).

6.3.4 Technology Mapping Based Optimizations

The XOR operator is not always useful in technology mapping. For instance, the Equation 6.4 implements a function with 6 literals using the XOR operator. Equation 6.4 can be synthesized as the Equation 6.5, using only AND/OR operations:

$$f = \bar{a} \cdot (\bar{b} \oplus ((\bar{b} \oplus \bar{c}) \cdot (\bar{b} \oplus \bar{d}))) \tag{6.4}$$

$$g = \bar{a} \cdot ((\bar{b} \cdot \bar{c}) + (\bar{d} \cdot (\bar{b} + \bar{c}))) \tag{6.5}$$

Equation 6.5 is the Boolean equivalent of the Equation 6.4 and also has 6 literals, using only AND/OR operations. Equation 6.4 contains 2 AND and 3 XOR operators, while in the Equation 6.5 have 3 AND and 2 OR operators. Since the XOR operator has a physical implementation cost higher than the AND/OR operators, the Equation 6.5 is preferred. The algorithm is capable of using costs and choosing the best implementation option. For instance, if the AND/OR operation is assigned an area cost of 2 and the XOR operation is assigned an area cost of 3, Equation 6.4 have a total cost equal 13 and Equation 6.5 have a total cost equal 10.

6.4 Example

An example to illustrate the algorithm execution is the function represented in following:

$$f = \bar{a} \cdot b \cdot d + a \cdot \bar{b} \cdot d + c \cdot d \tag{6.4}$$

After preprocessing (parse, constant allocation), the function is checked if is a read-once candidate, but this step fails (since the function is not read-once) and the algorithm computes the initial functions. This step computes the unateness information for variables. The variables ‘a’ and ‘b’ are binate and variables ‘c’ and ‘d’ are positive unate. There are two binate variables, and this allows the use of FC-EXACTXOR, instead of FC-EXACT. They are inserted in the 1-literal bucket.

The combination step is illustrated in Figure 6.2, which is a simplified version for a better figure legibility (each bucket has much more functions allocated). The 1-lit functions are combined producing the 2-literal functions. The combination process continues until the 4-literal bucket, where a solution is found.

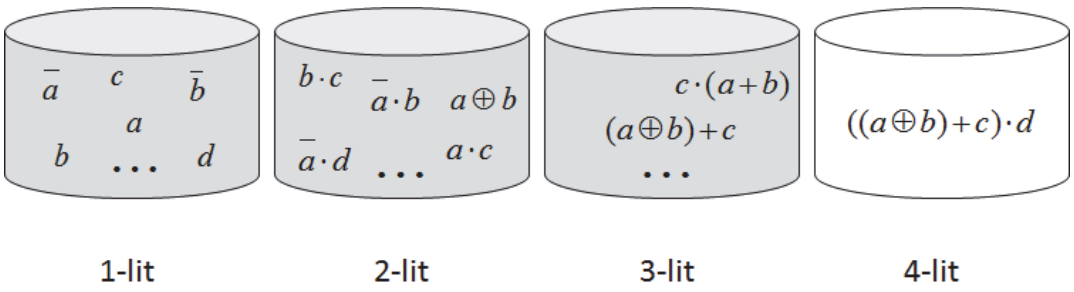


Figure 6.2: Combination step to find solutions for *f* using FC-EXACTXOR.

6.5 Experimental Results

Two experiments were executed to analyze the efficacy of the XOR factorization. The first experiment evaluates the impact of the XOR operator in a technology mapping and compares the results with the FC-EXACT. The second experiment analyzes the worst case execution time of FC-EXACT and compares with the execution time of FC-EXACTXOR. The experiments have been carried out at a computer with a Core2Duo 2.4 GHz processor and 4 GB RAM

The first experiment is a repetition of the experiment performed in Chapter 5, considering this time the FC-EXACTXOR factorization. The results are compiled in Table 6.4.

The use of the XOR operator by the FC-EXACTXOR produces exact factored forms with the set of operators {AND, OR, NOT, XOR}. This allows reducing the number of literals in 29% in comparison to the FC-EXACT, with 25405 literals. However, comparing literals with the XOR operation is not fair since the other expressions are factored with AND/OR. A fair analysis of the impact of XOR operator is made in technology mapping.

When performing technology mapping, the advantage of using the XOR operator in an expression, comparing the literal difference and the area difference is not proportional, because XOR physical implementation has a higher cost than AND/OR operators in commercial libraries. Even with a higher cost, the use of XOR operator provides an area reduction of 8%.

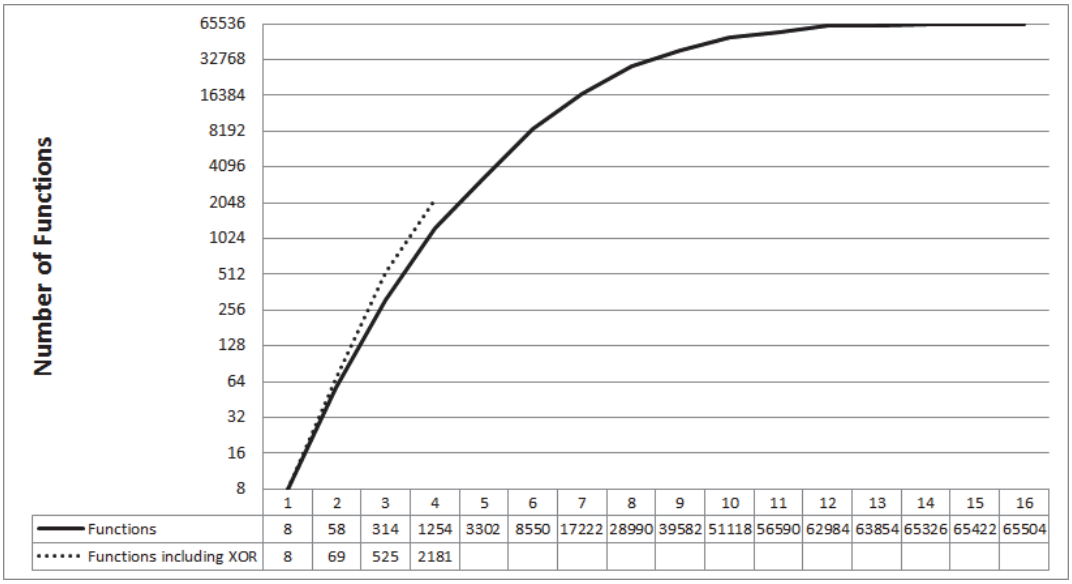
Table 6.4: Results comparing FC-EXACTXOR with other methods.

Method	Literals	Literal Difference over FC-EXACT	Area Difference over FC-EXACT	Execution time
FC-EXACT	36028	-	-	16h
FC-HEURISTIC	36738	+1.97%	+3.77%	114s
QF	38341	+6.42%	+6.61%	47s
GF	37893	+5.18%	+4.67%	47s
ABC	38246	+6.16%	+7.4%	2s
XF	37652	+4.51%	+1.1%	24s
FC-EXACTXOR	25405*	-29.49%*	-8.01%	40 min

*- These number cannot be fairly compared with the above numbers.

The computation time of factoring all 4-input functions (grouped in 3982 P-class functions) with FC-EXACTXOR was about 40 minutes, which is a great improvement of the FC-EXACT. To the best of the author knowledge, this is the first exact factoring algorithm that can minimize number of literals in factored forms considering the set of operators {AND, OR, NOT, XOR}.

The second experiment is an analysis of the number of functions allocated to factorize the XOR4 function in the FC-EXACT and FC-EXACTXOR. The number of functions computed in the FC-EXACT grows in an accelerated way. In this sense, it is needed to discard functions that do not aid in the search for the target function. The heuristics presented in Section 5.7.1 are not sufficient for this case. In Figure 6.3, it is illustrated this grow behavior (black line) using FC-EXACT for a function with 4 binate inputs (the XOR of 4 inputs). This function is represented by Equation 6.2. Almost all Boolean space for 4 inputs (2^{2^4} = 65536 functions) needs to be allocated to achieve the target function. In this way, it is important to reduce the number of intermediate functions (i.e. functions that are used to compose the target function). The dotted line represents the total number of functions using the FC-EXACTXOR algorithm, represented by the Equation 6.3. It is necessary only 4 buckets to find the solution, avoiding the generation of many more functions and speeding up the algorithm in almost 2 orders of magnitude. This means that the algorithm generates less than 2200 functions to achieve the solution. The execution time of the XOR4 function with FC-EXACT is 10 minutes and with FC-EXACTXOR is 2 seconds.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Functions	8	58	314	1254	3302	8550	17222	28990	39582	51118	56590	62984	63854	65326	65422	65504
Functions including XOR	8	69	525	2181												

Figure 6.3: XOR4 function distribution between buckets.

6.6 Conclusions

This chapter introduced an exact algorithm for factoring Boolean functions considering the set of operators {AND, OR, NOT, XOR}. We have demonstrated that exact minimization using XOR operator can produce a very significant reduction in the number of literals. We have also demonstrated that the gain obtained regarding number of literals produces area reduction after technology mapping, even if the XOR gate is

more expensive in commercial cell libraries when compared to AND/OR gates. To the best of the author knowledge, this is the first optimal factoring algorithm that is able to minimize number of literals in factored forms considering the set of operators {AND, OR, NOT, XOR},

In the context of technology mapping, almost all commercial libraries have at least a XOR2 cell. In this way, a possible and practical application generates a look-up table to optimize 4-input subcircuits, reducing even more the area, compared to FC-EXACT. This algorithm also can be multi-objective, allowing optimal minimization in multiple criteria, e.g. assigning costs to different operators; respecting MDC.

7 MAJORITY-BASED CIRCUIT SYNTHESIS

The advances in the field of IC digital design make possible the aggregation of an increasing number of devices on the same die. This high integration scale imposes new challenges to the synthesis process (ITRS). In an attempt to mitigate the increasing complexity of design of digital equipment's, a look is being taken at alternatives approaches to the problem. The analog and hybrid techniques do not provide the versatility of pure digital approach.

Threshold logic and its realization in the form of threshold gates offer a possible alternative to the Boolean approach. The threshold logic may allow a considerable economy in number of gates and interconnections necessary per circuit. Threshold gates are similar to normal Boolean gates in that their inputs and outputs are binary signals. The threshold gate is thus seen to be a logic function that can "weight" 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. In Figure 7.1 is shown a generic threshold logic gate.

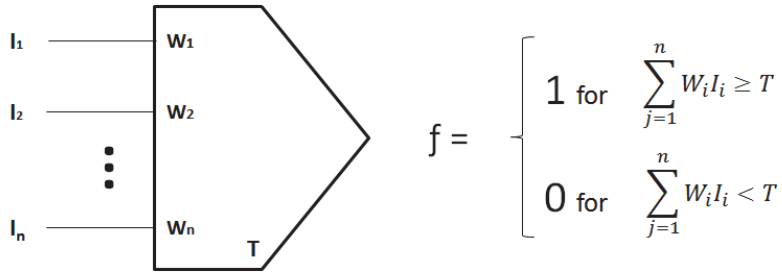


Figure 7.1: A n-input threshold logic gate.

A majority gate is a simplified version of a threshold gate, where the input weights have the same value and the output evaluate to '1' when more than half of inputs have logic '1', and the output evaluates to '0', otherwise. A minority gate is the complemented version of a majority gate.

7.1 Related Work

Threshold logic synthesis' research dates back to 1960s. Akers (AKERS, 1962), Miller and Winder (MILLER; WINDER, 1962), and Muroga (MUROGA, 1971) employed logic synthesis methods based on the reduced-unitized table, Karnaugh map, and Shannon's decomposition principles, respectively. Traditional logic reduction methods, such as ESPRESSO (BRAYTON ET AL, 1984), always produce simplified expressions in the two standard forms: SOP or POS. However, difficulties are found in

converting SOP/POS forms into majority circuits due to the complexity of multilevel majority gates.

Since the synthesis analyzed in this chapter is based on a majority gate primitive, it is critical that an efficient technique is established for designing with the majority gate primitive. However, these methods have a common drawback that they are only suitable for synthesizing small networks manually. Recently, some majority logic reduction methods targeting quantum cellular automata (QCA) (LENT ET AL, 1993; LENT; TOUGAW, 1997), tunneling phase logic (TPL) (FALUNY; KIEHL, 1999) and single electron tunneling (SET) (AVERIN**;** LIKHAREV, 1986) circuits have been proposed. All these technologies use majority or minority gates as primitive elements.

Rumi Zhang in (ZHANG ET AL, 2004) pointed out a set of 13 functions of 3 variables implemented using only majority gates and inverters. This set is also the set of functions called 3-NPN. An NPN set is a class of functions equivalent to each other, considering the **P**ermutation of its inputs, complementation (**N**egation) of its inputs, and/or inversion (**N**egation) of its output. This set aims to reduce the hardware requirements for a QCA design, working as a cell library, but these 13 functions are not in minimal form, i.e., a minimal number of majority gates.

Rui Zhang in (ZHANG; GUPTA; JHA, 2005) proposed a different flow using factoring algorithms to synthesize functions, instead of using a cell library and this algorithm allows majority and minority circuit synthesis. However, the algorithm most of the time convert the AND/OR logic gates from a factored expression to majority gates, which can negatively impact in the total number of majority gates in a circuit.

Momenzadeh in (MOMENZADEH ET AL, 2005) optimized two functions of (ZHANG ET AL, 2004) that were not implemented in a minimal number of majority gates and proposed an And-Or-Inverter (AOI) structure composed of 2 majority gates connected in series to reduce even more the number of majority gates present n a circuit.

Kong (KONG; YUN; LU, 2010) improved (ZHANG; GUPTA; JHA, 2005), ensuring and proving optimality for three variable functions. The library proposed was expanded to 40 functions, reducing the area considerably.

However, these methods only support three variable Boolean functions. To synthesize arbitrary multi-variable Boolean functions, a QCA majority synthesis methodology was introduced in (ZHANG; GUPTA; JHA, 2005), decomposing a circuit to subcircuits with 3 or fewer inputs. In majority logic synthesis, there still exist some important aspects that are not solved or considered by the existing methods. (AKERS, 1962; MILLER; WINDER, 1962;MUROGA, 1971; ZHANG ET AL, 2004; ZHANG; GUPTA; JHA, 2005; MOMENZADEH ET AL, 2005; KONG; YUN; LU, 2010). These methods are not capable of generating optimal structures with majority gates with more than 3 variables. The 4-NPN function class has 222 functions, and the 4-P function class has 3984 functions, making the generation of these libraries unfeasible without computational aid. For this reason, it is important an automated method who can synthesize more than 3 inputs.

As seen in the Chapter 5 and Chapter 6, the FC-EXACT and FC-EXACTXOR can generate expressions with a minimal number of literals. The flexibility of FC allows taking advantage of bonded-pairs complex association to generate structures with only majority gates and inverters and exploring optimality of two criterions: logic depth and

a number of majority gates. Table 7.1 compares the previous works and the majority-based circuit synthesis algorithm using FC, named herein as FC-MAJ (MARTINS ET AL, 2012). The column "majority generation" is how the algorithm performs the circuit, or by real-time synthesis or using a library to perform technology mapping.

Table 7.1: Comparison between majority-based synthesis algorithms

Method	Exactness	Max inputs	Templates allowed	Majority Generation
(ZHANG ET AL, 2004)	Heuristic	3-input	MAJ	Library
(MOMENZADEH ET AL, 2005)	Exact	3-input	MAJ,AOI	Library
(ZHANG; GUPTA; JHA, 2005)	Heuristic	3-input	MAJ,MIN	Synthesis
(KONG; SHANG; LU, 2010)	Exact	3-input	MAJ, MIN	Synthesis
FC-MAJ(MARTINS ET AL, 2012)	Exact	4-input	Any	Library

In this chapter, majority function will represent a 3-input majority gate output function. The optimal factored form of a majority function (MF) is expressed in Equation 7.1.

$$maj(a,b,c) = a \cdot (b + c) + b \cdot c \tag{7.1}$$

A method to compose an MF using Boolean functions needs four logic operations, the same number of operators in the Equation 7.1. Since all variables of MF are positive unate, there is the necessity of inverters to represent negative unate and binate functions. The MF is symmetric, thus changing the order of inputs does not change the logic function.

Another interesting property of an MF is to be a self-dual function. This property allows easy conversion from majority-based circuits to minority-based circuits. If a majority gate has the output negated, the majority gate acts as a minority gate. If the minority gate has the inputs complemented, the minority gate act as majority gate as seen in Figure 7.2.

Figure 7.2: Self-dual property in majority gates.

Considering all MF properties, the bonded-pair association needs to be performed using three bonded-pairs. The circuit structure stores the majority gates and its connections. This information is important, since it allows a traverse backward in the structure, counting all majority gates forming the circuit that implements the function.

7.2 Functional Composition Setup for Majority Gate Circuit Synthesis

In this section, the FC setup for the synthesis of majority gates is presented. All choices are explained in the further sections.

1. Bonded-Pair Representation: {function, majority gate based circuit}
2. Initial Functions: Variables, constant 0 and constant 1

3. Bonded-Pair Association: Complex association (3 functions to generate a new one)
4. Partial Order: Logic Depth/Majority gate count
5. Allowed Functions: All functions of up to *n* variables

7.3 Partial Order Criterion

There are criteria for partial order when synthesizing majority gates based circuits using FC. One is using the logic depth. Logic depth is the maximum number of gates a signal needs to travel from the input to output. The logic depth is related to the delay of a logic gate. Another criterion is using the number of majority gates representing each function. Reducing the number of majority gates impacts directly in the final area of the circuit.

7.3.1 Number of Majority Gates Approach

This approach uses a number of majority gates as a primary criterion and logic depth as a secondary criterion. Each bucket is classified by the minimal amount of majority gates needed to represent a Boolean function. If two functions are implemented with the same number of majority gates, the function with minimal logic depth is chosen.

For the initial functions, only variables in both polarities (complemented and uncomplemented) and constants are included. This initial set will be called 0-maj, since majority gates are not necessary to represent variables in positive or negative form.

To compose an *n*-maj bucket, three functions are needed (and their respective implementations) from previous buckets that respect the Equation 7.2.

$$n = M_1 + M_2 + M_3 - 1 \tag{7.2}$$

In Equation 7.2 M_1, M_2, M_3 are the number of majority gates in the circuits representing the functions that were used to compose the new function. These circuits are connected to compose a new majority gate circuit.

Example 1: To compose a 5-maj bucket, it needs the combination of the previous buckets indicated in Table 7.2. I_1, I_2 and I_3 represent the inputs of a majority gate. A 5-maj function can be composed of a function which is represented by a circuit with 4 majority gates and two functions that do not needs majority gates (variables and constants). The other lines in the table represent the other possible combinations. The symmetry in the majority function reduces the cases (e.g. the case 4-0-0 is equal to case 0-4-0, so the latter one is discarded.)

Example 2: The combinations to compose the 3-maj bucket are shown in Table 7.2 and the approach is depicted in Figure 7.3.

Table 7.2: Possible combinations to create 5-maj bucket.

I_1	I_2	I_3
4-maj	0-maj	0-maj
3-maj	1-maj	0-maj
2-maj	2-maj	0-maj

2-maj 1-maj 1-maj

Table 7.3: Possible combinations to create a 3-maj bucket.

I_1	I_2	I_3
2-maj	0-maj	0-maj
1-maj	1-maj	0-maj

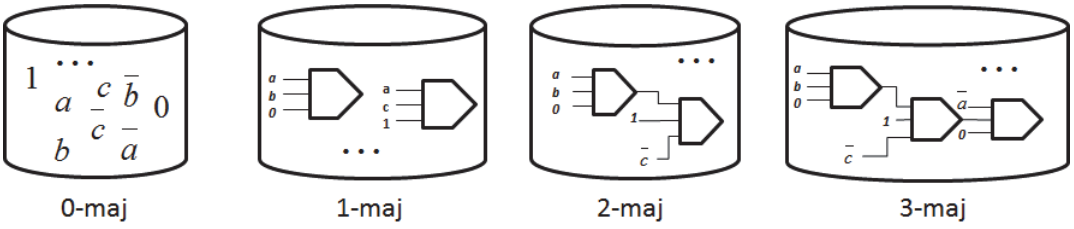


Figure 7.3: Number of majority gate approach example.

7.3.2 Logic Depth Approach

This approach considers the logic depth minimization. If two functions are implemented with the same logic depth, the function with fewer majority gates is preferred. Note that the maximum number of majority gates (M) in a *k*-depth is given by the Equation 7.3:

$$M = \frac{3^{k+1} - 1}{2} \tag{7.3}$$

The Equation 7.3 is derived from a geometric series with common ratio equal three. This ratio represents the number of inputs of a majority gate.

For the initial functions, only variables in both polarities and constants are included. This initial set will be called 0-depth, since majority gates are not necessary to represent variables in positive or negative form, as well constants.

The 1-depth contains all functions that can be synthesized with 1 majority gate and they are in the optimal form. The 1-depth elements are formed by the combination, 3 by 3, of the elements contained in the 0-bucket.

Lemma 7.1: In order compose a n-depth bucket, at least one majority gate composed in (n-1)-depth connected in a input is necessary. The other inputs can be from the any i-depth, $0 \leq i < n$.

Consider a function *f* that is implemented as a majority gate circuit with the minimal logic depth *n*. Consider a set G_0 containing all functions that are implemented as majority gate circuits with minimal logic depth, up to *n-2*. Consider a set G_1 containing all functions that are implemented as majority gate circuits with minimal logic depth, equal *n-1*. By Lemma 7.1, the implementation of *f* is one of three cases, illustrated in Figure 7.4 and listed below:

1. The inputs of the *n*-depth majority gate are one structure from G_1 set and the other two from G_0 set.

2. The inputs of the n-depth majority gate are two structures from the G_1 set and the other one from the G_0 set.
3. The inputs of the n-depth majority gate are the three structures from the G_1 set.

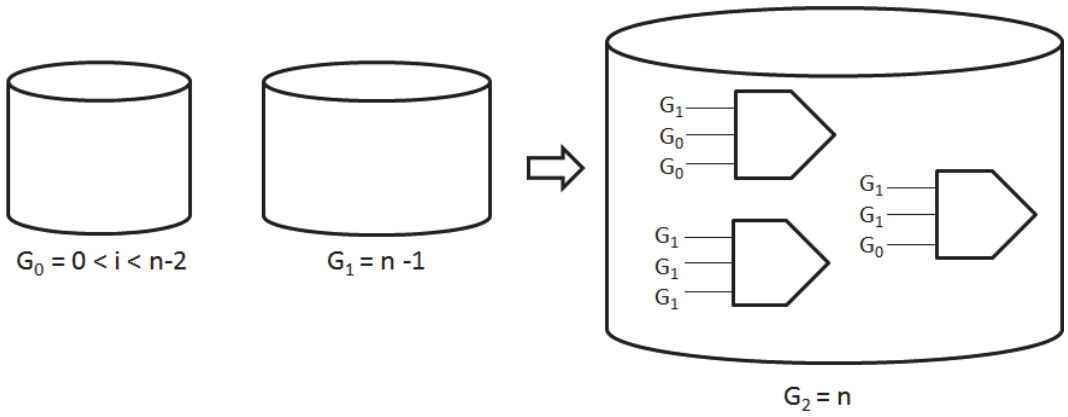


Figure 7.4: Generating a n-depth bucket.

Example: To compose a 3-depth bucket, the combination of the previous buckets indicated in Table 7.4 is necessary. A 3-depth function can be composed of a function which is represented by a circuit with at least logic depth equals two. Figure 7.5 shows the logic depth approach.

Table 7.4: Possible combinations to create a 3-depth bucket.

I_1	I_2	I_3
2-depth	0-depth	0-depth
2-depth	1-depth	0-depth
2-depth	1-depth	1-depth
2- depth	2- depth	0- depth
2- depth	2- depth	1- depth
2- depth	2- depth	2- depth

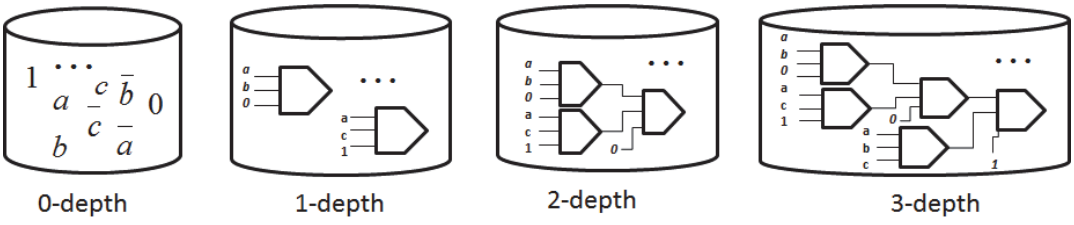


Figure 7.5: Logic depth approach.

7.4 Inverter Cost

The number of inverters present in each circuit is very important, especially in technologies as QCA In the QCA technology, the inverter has almost the double area

than the majority gate, according to (MOMENZADEH ET AL, 2005). In this sense, there is a necessity of a post-optimization in the number of inverters.

It is possible to exploit the self-dual property of majority gates, to reduce the number of inverters in the circuit. As seen in Figure 7.6, the circled cases have the minimal inverter count in a majority gate, except for the third case, which has the same number of inverters in its equivalent representation.

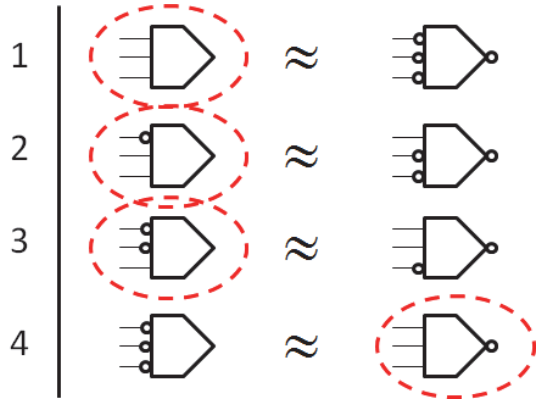


Figure 7.6: Exploring the self-dual property to reduce the number of inverters.

7.5 Synthesizing a Library

A library is a finite set of primitive logic gates, including combinational, sequential (e.g. flip-flops) and interface (e.g. drivers) elements. The interest in this chapter is only the combinational part, where each element implements a Boolean function. It is interesting to have the maximum number of functions, allowing flexibility in the technology mapping. The majority gate based library is composed of cells with majority gates and inverters as a primitive structure to represent the functions.

Using FC principles, the partial order used in this chapter is the logic depth. Each depth is generated in order to have each function represented by the optimal structure (the minimum number of majority gates and inverters). The process combines initially functions with a smaller cost (i.e., fewer majority gates) to guarantee the optimal results. If a function is generated and already exists (i.e. was generated with fewer depth and gates), this function is discarded. The inverters are optimized, using the technique shown in Section 7.4.

Example: In Figure 7.7, the generation of a library with 2 inputs is shown. In the 0-depth are allocated all variables in the positive and negative polarity and the constants. In the 1-depth, there are all functions that can be synthesized with 1 majority gate. In the 2-depth, the light gray majority gates from the 1-depth ($\bar{a}\cdot\bar{b}, a\cdot b$) 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 ($a\cdot\bar{b}, \bar{a}\cdot b$) 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.

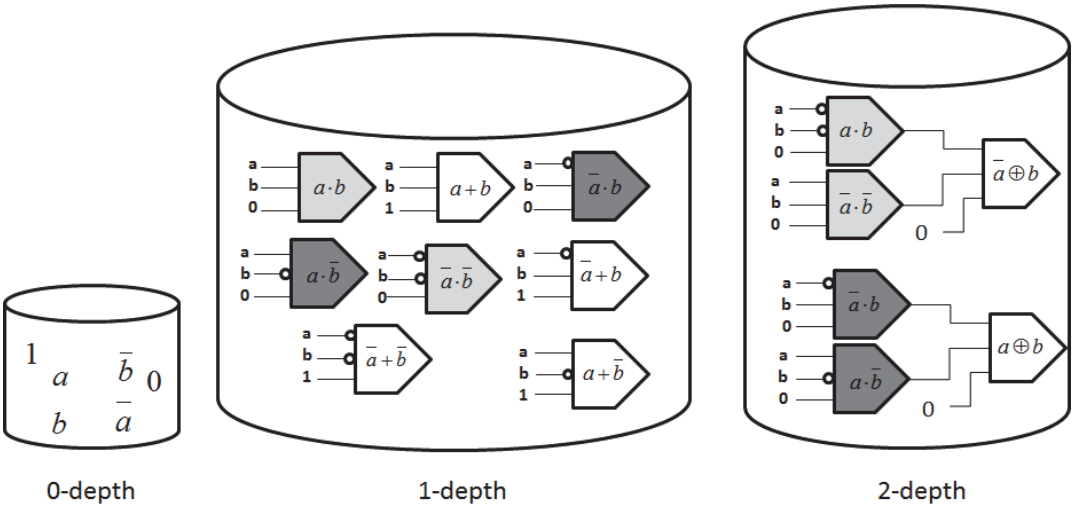


Figure 7.7: Generation of all functions up to 2 variables.

7.6 Experimental Results

Two experiments were performed. The first experiment generated all 3-input functions. These functions were grouped into two subsets, one subset with 80 P-class functions and the other subset of 13 NPN-class functions). These sets are mapped using the FC-MAJ (MARTINS ET AL, 2012), applying inverter post-optimization and are compared with (KONG; SHANG; LU, 2010) in Table 7.5 .

Table 7.5: Results (in number of majority gates) of FC-MAJ for 3-input functions.

Method	MAJ (3NPN)	INV (3NPN)	MAJ (3P)	INV (3P)
FC-MAJ (MARTINS ET AL, 2012)	35	24	218	152
(KONG; SHANG; LU, 2010)	35	24	-	-

The algorithm described in (KONG; SHANG; LU, 2010) guarantees minimal majority gate count results for 3 variables and the FC-MAJ algorithm achieved the same results. Another result is the synthesis of the 3-P class functions. EDA tools uses P-matching algorithms (DEBNATH; SASAO, 1999; MARTINELLO ET AL, 2010) in the technology mapping, thus the importance of a library containing functions representing unique P-classes.

The second experiment generated all 4-input functions. These functions were grouped into two subsets, one subset with 3984 P-class functions and the other subset with 222 NPN-class functions, as shown in Table 7.6.

Table 7.6: Results (in number of majority gates) of FC-MAJ for 4-input functions.

Method	MAJ (4NPN)	INV (4NPN)	MAJ (4P)	INV (4P)
FC-MAJ	1739	233	32010	5076

This is the first algorithm to synthesize 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 will be implemented in a future work to compare an optimal logic depth library and an optimal majority gate count library.

The distribution of the majority gates in the 4-P and 4-NPN function classes is shown in Figure 7.8. The distribution of logic depth in the 4-P and 4-NPN is shown in Figure 7.9. The histograms are shown in log scale for better visualization. It is worth to mention two important details. One is that only two functions are implemented with 4-depth, the XOR4 and XNOR4. Surprisingly, these functions are not the most complex in number of majority gates, needing only 9 majority gates. Second detail is the absence of 4-input functions needing 10 majority gates. Indeed, the upper bound majority gate count for 3-depth is 13, by Equation 7.3. All the functions (except XOR4 and XNOR4) can be synthesized with 4-depth or less. Almost all functions up to 4 inputs can be implemented with majority gate count equal 9 or less.

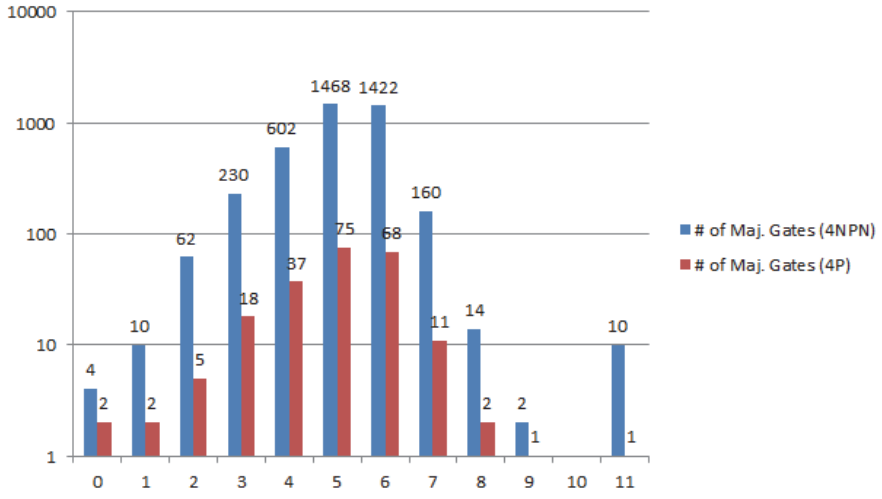


Figure 7.8: Histogram for the 4-input library, considering the number of majority gates to implement the functions.

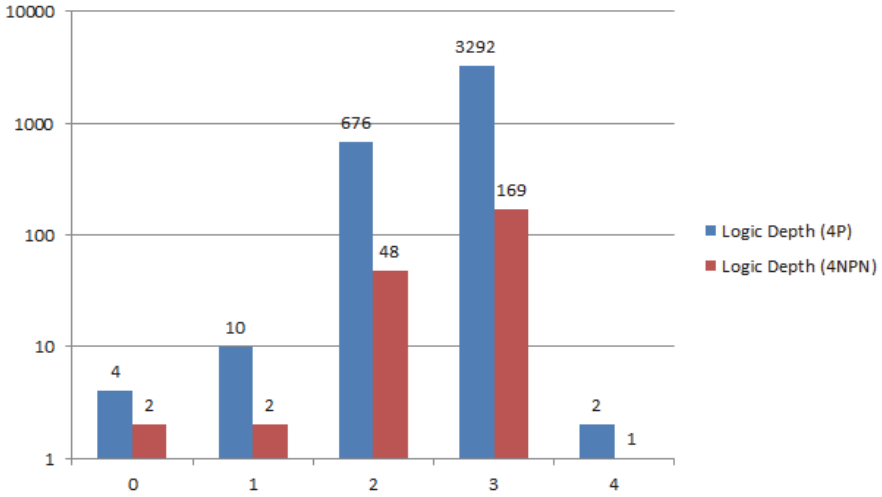


Figure 7.9: Histogram for 4-input library, considering the logic depth of the functions.

7.7 Conclusions

In this chapter, an algorithm was introduced for synthesizing circuits using only majority gates and inverters, suitable for use in new technologies, as QCA, SET and TPL. FC-MAJ generates the optimal structure of majority gates, given a function and can generate libraries in an automated way, using the FC paradigm. All techniques in the literature, to the best knowledge of the authors, can only handle 3-input functions and the FC-MAJ can handle 4-inputs, which can reduce the area of the circuit considerably.

8 CONCLUSIONS AND FUTURE WORK

The main contribution of this work is the introduction of a novel paradigm for performing logic synthesis, called as functional composition. It is based on a bottom-up approach exploiting composition of Boolean functions to have an efficient cost control. Four applications methods have been presented with promising results, demonstrating the potential and allowing space for implementation of new algorithms using the functional composition paradigm.

The first application is the FC-MDC, which is an efficient method to compute minimum decision chains (MDC) of logic functions. The FC-MDC method is compared to the QMC-MDC method, which is a modified version of the Quine-McCluskey algorithm for MDC computation. The QMC-MDC method presents some limitations related to the number of prime implicants of the functions, as it has to compute nearly all prime implicants, making it quite computing expensive in some cases. The FC-MDC method is a complementary method of QMC-MDC and is faster especially in the cases of CMOS design interest, i.e., logic functions with MDC smaller than 5.

The second application presents a Boolean factoring algorithm, divided into two approaches. The heuristic approach (FC-HEURISTIC) is the first multi-objective factoring algorithm. From a quality point of view, the proposed algorithm always delivered superior (or equal) results compared to other approaches. The algorithm can take a secondary criterion (like series and parallel number of switches, or logic depth) into account while generating several alternative solutions. This characteristic makes the FC-HEURISTIC a useful piece for approaches based on restructuring small portions of logic, like (WERBER; RAUTENBACH; SZEGEDY, 2006) and (MISHCHENCKO; BRAYTON; CHATTERJEE, 2008). The unique characteristics of the algorithm make it very useful in the context of local optimizations. The other approach is an exact algorithm (FC-EXACT) for factoring Boolean functions. This approach provides minimal results in number of literals. One interesting application is factoring a set of functions and storing them in a look-up table, for use in a synthesis flow.

The third application is the FC-EXACTXOR, an exact algorithm for factoring Boolean functions using all operators in the set {AND, OR, NOT, XOR}, providing a minimal form with all these operators. The FC-EXACTXOR is capable of considering different costs to AND/OR/XOR operators. The gain obtained regarding number of literals produce area reduction after technology mapping, even if the XOR cell is more expensive in commercial cell libraries compared to AND/OR/NAND/NOR cells.

The fourth application is an algorithm for synthesizing circuits using only majority gates and inverters, suitable for quantum cellular automata (QCA) design, tunneling phase logic (TPL), single electron tunneling (SET) and other technologies that use majority and minority gates as primitive structures. The application, called FC-MAJ,

generates the optimal structure of majority gates, for a given a function and it can generate libraries in an automated way, using the functional composition paradigm. All techniques in the literature, to the best of author knowledge, can only handle 3-input functions, while the FC-MAJ in this thesis achieved optimal results in logic depth for 4-input functions.

There is still much work that needs to be carried out. Other applications can exploit the FC principles and generate better results than known algorithms. The heuristic factoring algorithms can be optimized, taking advantage from other methods to generate better initial bonded-pairs, providing better and faster results, as the disjoint support decomposition (BERTACCO; DAMIANI, 1997). The implementation of the FC-MAJ using number of majority gates as partial order needs to be implemented.

REFERENCES

AKERS, S. B.. Binary Decision Diagram**s. IEEE Transactions on Computers.** 27, 6 , 509-516. June 1978.

AKERS, S. B.. Synthesis of combinational logic using three-input majority gates, In: 3rd Annual Symposium Switching Circuit Theory and Logical Design, **Proceedings** of, pp. 149–157, Oct. 1962

ASHENHURST, R. L.. The decomposition of switching functions. In: International Symposium on the Theory of Switching, **Annals** of, Harvard University, pp. 74-116, vol. 29, 1959

AVERIN, D. V.; LIKHAREV, K. K.. Coulomb blockade of single-electron tunneling, and coherent oscillations in small tunnel junctions**, Journal of Low Temperature Physics**, vol. 62, pp. 345-373, Fev. 1986.

Berkeley Logic Synthesis and Verification Group, **ABC: A System for Sequential Synthesis and Verification**, Release 051205. http://www.eecs.berkeley.edu/~alanmi/abc/

BERTACCO, V.; DAMIANI, M.. The disjunctive decomposition of logic functions. In: The 1997 IEEE/ACM international conference on Computer-aided design (ICCAD '97), **Proceedings** of, Washington, DC, USA, 78-82. 1997.

BOOLE, G.. **An Investigation of the Laws of Thought**. Walton, London, 1854. (reprinted by Dover, New York, 1958).

BRAYTON, R. K.. Factoring logic functions. **IBM Journal of Research and Development**, vol. 31, no. 2, pp.187-98. Mar 1987.

BRAYTON, R. K.; SANGIOVANNI-VINCENTELLI, A. L.; MCMULLEN, C. T.; HACHTEL, G. D.. **Logic Minimization Algorithms for VLSI Synthesis.** Kluwer Academic Publishers, Norwell, MA, USA. 1984.

BRYANT, R. E.. Graph-based algorithms for Boolean function manipulation. **IEEE Transactions on Computers**, vol. C-35, no. 8, pp.677-691. Aug. 1986.

CARUSO, G.. An improved algorithm for Boolean factoring, In: IEEE International Symposium on Circuits and Systems, **Proceedings** of, pp.241-244 vol.1, Jun 1994.

CHATTERJEE, S.; MISHCHENKO A.; BRAYTON, R.; WANG, X; KAM, T.. Reducing structural bias in technology mapping, In: International Conference on Computer-Aided Design, **Proceedings** of, pp.519,526, 6-10 Nov. 2005

CORREIA, V.; REIS, A.. Advanced technology mapping for standard-cell generators, Symposium on Integrated Circuits and Systems Design, **Proceedings** of**,** pp. 254- 259, 2004.

CORREIA, V.P.; REIS, A.I.. Classifying n-Input Boolean Functions. IBERCHIP Workshop, **Proceedings** of, pp. 58-66, 2001.

COUDERT, 0.. Two-level logic minimization: an overview. **The VLSI journal Integration,** 17-2, pp. 97- 140, Oct. 1994

CURTIS, H. A.. **A New Approach to the Design of Switching Circuits**. Von Nostrand, 1962.

DA ROSA, L. S. JR.; MARQUES, F. S.; CARDOSO, T. M. G.; RIBAS, R. P.; SAPATNEKAR, S. S.; REIS, A. I.. Fast disjoint transistor networks from BDDs, In: 19th Annual Symposium on Integrated Circuits System Design, **Proceedings** of, pp.137 2006

DEBNATH, D.; SASAO, T.. Fast Boolean Matching Under Permutation Using Representatives, In: Asia and South Pacific Design Automation, **Proceedings** of, pp. 359–362, 1999.

FALUNY, H.A.H.; KIEHL, R.A.. Complete logic family using tunneling-phase-logic devices, In: The Eleventh International Conference on Microelectronics, **Proceedings** of, pp. 153- 156, 22-24 Nov. 1999

FIGUEIRO, T.; RIBAS, R. P.; REIS, A. I.. Constructive AIG optimization considering input weights, In: International Symposium on Quality Electronic Design, **Proceedings of,** pp. 1-8, Mar. 2011

GAJSKI, D.D.; KHUN, R.H.. **Guest Editors' Introduction: New VLSI Tools** Computer , vol.16, no.12, pp.11-14, Dec. 1983

GOLUMBIC, M. C.; MINTZ, A.. ROTICS, U.; An improvement on the complexity of factoring read-once Boolean functions. **Discrete Applied Mathematics** , vol. 156, no. 10, pp.1633-36, May 2008

GOLUMBIC, M. C.; MINTZ, A.; ROTICS, U.. Factoring and recognition of read-once functions using cographs and normality**.** In: The 34th Annual Design Automation Conference (DAC), **Proceedings** of, pp.109-14, 2001.

GOLUMBIC, M. C.; MINTZ, A.. Factoring logic functions using graph partitioning. In: IEEE/ACM International Conference on Computer-Aided Design, **Proceedings** of pp.195-199, 1999

GRAY, F.; **Pulse code communication**. U.S. Patent 2,632,058; Mar 1953

HACHTEL, G. D.; SOMENZI, F.. **Logic Synthesis and Verification Algorithms**, Springer, 2006. 564p.

HASSOUN, S.; SASAO, T. (Ed.). **Logic Synthesis and Verification**. Norwell, MA, USA: Kluwer Academic Publishers, 2002

HLAVICKA, J.; FISER, P.. BOOM-a heuristic Boolean minimizer, In: IEEE/ACM International Conference on Computer Aided Design, **Proceedings** of, pp.439,442, 4-8 Nov. 2001

International Technology Roadmap for Semiconductors (ITRS). http://www.itrs.net

JOZWIAK, L.; BIEGANSKI, S.. Technology Library Modelling for Information-driven Circuit Synthesis, In: 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools, **Proceedings** of, pp.480,489, 3-5 Sept. 2008

JOZWIAK, L.. Information relationships and measures in application to logic design, In: 29th IEEE International Symposium on Multiple-Valued Logic, **Proceedings** of, pp.228-235, 1999.

KAGARIS, D.; HANIOTAKIS T.. A Methodology for Transistor-Efficient Supergate Design, **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, vol.15, no.4, pp.488,492, April 2007

KARNAUGH, M.. The Map Method for Synthesis of Combinational Logic Circuits. **Transactions of the American Institute of Electrical Engineers**, pp. 593–599, Nov 1953.

KARPLUS, K.. **Using if-then-else DAG's for multi-level logic minimization;** Univ. California, Santa Cruz, UCSC-CRL-88-29, 1988.

KILBY, J.. **Semiconductor Structure Fabrication**, U.S. Patent 3,072,832 filed May 1959, issued January 1963

KONG, K.; SHANG, Y.; RUQIAN, L.. An Optimized Majority Logic Synthesis Methodology for Quantum-Dot Cellular Automata, **IEEE Transactions on Nanotechnology**, vol.9, no.2, pp.170-183, 2010

KONG, J.; HUSSAIN, S. Z.; OVERHAUSER, D.. Performance estimation of complex MOS gates**, IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, ,** vol.44, no.9, 785-795, Sep 1997.

LAWLER, E. L. An approach to multilevel Boolean minimization, **Journal of ACM**, vol.11, no.3, July 1964. pp. 283-95.

LEE, C. Y.. Representation of switching circuits by binary-decision programs. Bell **System Technology Journal**, 985-999; Jul 1959.

LEE, T.. WANG, C. Recognition of fanout-free functions. In: Asia and South Pacific Design Automation Conference (ASP-DAC), **Proceedings** of pp.426-31. Jan. 2007.

LENT, C.. TOUGAW, P.; A device architecture for computing with quantum dots, **Proceedings of the IEEE**, vol.85, no.4, pp.541,557, Apr 1997.

LENT, C. S.; TOUGAW, P. D.; POROD, W.; BERNSTEIN, G. H.. Quantum cellular automata, **Nanotechnology,** vol 4 pp.49, 1993.

LIU, Y.; SHELAR, R.S.; HU, J.. Simultaneous Technology Mapping and Placement for Delay Minimization, **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,** vol.30, no.3, pp.416-426, March 2011.

MARKOV, S. J.; SYLVESTER, I. L.; BLAAUW, D.. On the decreasing significance of large standard cells in technology mapping. In: 2008 IEEE/ACM international Conference on Computer-Aided Design (ICCAD'08), **Proceedings** of, pp.116-121, 10, Nov 2008.

MARQUES, F.S.; ROSA, Jr. L.S.; RIBAS, R.P.; SAPATNEKAR S.S.; REIS, A.I.. DAG based library-free technology mapping., In: Great Lakes Symposium on VLSI, **Proceedings** of, pp. 293-298, 2007.

MARTINELLO Jr., O.; MARQUES, F. S; RIBAS, R. P.; REIS, A. I.. KL-cuts: a new approach for logic synthesis targeting multiple output blocks. In: Conference on Design, Automation and Test in Europe (DATE), **Proceedings** of, pp.777-82. 2010.

MARTINS, M. G. A.; ROSA JR, L. S.; RASMUSSEN, A. B.; RIBAS, R. P.; REIS, A. I.. Boolean factoring with multi-objective goals, In: IEEE International Conference on Computer Design (ICCD), **Proceedings** of, pp.229,234, 2010

MARTINS, M.G.A.; CALLEGARO, V.; RIBAS, R. P.; REIS, A. I.. Efficient method to compute minimum decision chains of Boolean functions, In: 21st edition of the great lakes symposium on Great lakes symposium on VLSI (GLSVLSI). **Proceedings** of, pp. 419-422, 2011.

MARTINS, M. G. A.; CALLEGARO, V; RIBAS, R. P.; REIS, A. I.; Computing Minimum Decision Chains of Boolean Functions. In: 26th South Symposium of Microelectronics, **Proceedings** of, 2011.

MARTINS, M. G. A.; RIBAS, R. P.; REIS, A. I.; Functional composition: A new paradigm for performing logic synthesis. In: International Symposium on Quality Electronic Design (ISQED), **Proceedings** of, 2012.

MARTINS, M. G. A.; RIBAS, R. P.; REIS, A. I.; Applications of Functional Composition. In: 27th South Symposium of Microelectronics, **Proceedings** of, 2012.

MARTINS, M. G. A.; CALLEGARO, V; MACHADO, L.; RIBAS, R. P.; REIS, A. I.; Functional Composition Paradigm and Applications. In: International Workshop on Logic and Synthesis, **Proceedings** of, 2012.

MCCLUSKEY, E. J.. Minimization of Boolean functions, **The Bell System Tech. Journal**, vol.35, no.5, Nov.1956

MCGEER, P.C.; et al.. ESPRESSO-SIGNATURE: a new exact minimizer for logic functions, **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, vol.1, no.4, pp.432,440, Dec. 1993

MICHELI, G. D. **Synthesis and Optimization of Digital Circuits**. McGraw-Hill Higher Education, 1994.

MILLER, H. S. ;WINDER, R. O..Majority-Logic Synthesis by Geometric Methods, **IRE Transactions on Electronic Computers,** , vol.EC-11, no.1, pp.89,90, Feb. 1962.

MINTZ, A; GOLUMBIC, M. C.. Factoring Boolean functions using graph partitioning. **Discrete Applied Mathematics**, vol. 149, no. 1–3. pp.131-53. 2005.

MISHCHENKO, A.; BRAYTON, R.; CHATTERJEE, S.. Boolean factoring and decomposition of logic networks, In: Computer-Aided Design, IEEE/ACM International Conference on, **Proceedings** of, pp.38,44, 2008.

MISHCHENKO, A.; BRAYTON, R.. JANG, S.; KRAVETS, V.; Delay optimization using SOP balancing, In: International Workshop on Logic and Synthesis (IWLS'2011), **Proceedings** of, 2011.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R.. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis, In: 43rd ACM/IEEE Design Automation Conference, **Proceedings** of, pp.532,535, 2006.

MISHCHENKO, A.; CHATTERJEE, S.; JIANG, R.; BRAYTON, R.. FRAIGs: A Unifying Representation for Logic Synthesis and Verification, **ERL Technical Report**, EECS Dept., UC Berkeley, March 2005.

MOMENZADEH, M.; HUANG, J.; TAHOORI, M. B.; LOMBARDI, F.. Characterization, test, and logic synthesis of and-or-inverter (AOI) gate design for QCA implementation, **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, vol.24, no.12, pp. 1881- 1893, 2005.

MOORE, G. E.. Cramming more components onto integrated circuits. **Electronics Magazine**. 4-8. Apr 1965.

MUROGA, S.. **Threshold logic and its applications.** Wiley Interscience**,** New York, 1971.

NGUYEN, L.. PERKOWSKI, M.; GOLDSTEIN, N.; PALMINI-Fast Boolean Minimizer for Personal Computer, In: 24th Conference on Design Automation, **Proceedings** of, pp.615,621, 1987.

PERKOWSKI, M. A.; CSANKY L.; SARABI, A.; SCHAFER I.. Fast minimization of mixed-polarity AND/XOR canonical networks, In: IEEE 1992 International Conference on Computer Design: VLSI in Computers and Processors, **Proceedings** of, pp.33,36, 11-14 Oct 1992

PLAZA, S.; BERTACCO, V.. STACCATO: disjoint support decompositions from BDDs through symbolic kernels, In: Asia and South Pacific Design Automation Conference, **Proceedings** of, vol.1, pp. 276- 279, 2005

REIS, A.I.. Covering strategies for library free technology mapping., In: XIIth conference on Integrated circuits and systems design, Proceedings pp. 180-183,

REIS, A. I.; ROBERT, M.; REIS, R..Topological parameters for library free technology mapping, In: XI Brazilian Symposium on Integrated Circuit Design**, Proceedings**. of, pp.213-216, 30 Sep-3 Oct 1998

RUDELL, R.L.: Logic Synthesis for VLSI Design, **PhD Thesis**, UCB/ERL M89/49,1989

ROYCHOWDHURY, V. P.; SIU, K.; ORLISTSKY, A.; **Theoretical Advances in Neural Computation and Learning**. Kluwer Academic Publishers, Norwell, MA, USA, 1994

SASAO, T.; A Design Method for AND-OR-EXOR Three-Level Networks, In: International Workshop on Logic and Synthesis, **Proceedings** of, 2005

SASAO, T.; HAMACHI, I.; WADA, S.; MUNEHIRO, M.. Multi-level Logic Synthesis Based on Pseudo-Kronecker Decision Diagrams and Local Transformation, In: Reed-Muller Workshop, **Proceedings** of, pp. 152-160, 1995.

SCHNEIDER, F.R.; REIS, A.I.; RIBAS, R.P.; 24th Norchip Conference, **Proceedings** of, pp.85,88, Nov. 2006

SCHNEIDER, F. R.; RIBAS, R. P.; SAPATNEKAR, S. S.; REIS, A. I.; Exact lower bound for the number of switches in series to implement a combinational logic cell, IEEE International Conference on Computer Design: VLSI in Computers and Processors, **Proceedings** of, pp.357,362, 2-5 Oct. 2005

SENTOVICH, E.; SINGH, K.; LAVAGNO, L.; MOON, C.; MURGAI, R.; SALDANHA, A.; SAVOJ, H.; STEPHAN, P.; BRAYTON, R.; SANGIOVANNI-VINCENTELLI, A.. **SIS: A system for sequential circuit synthesis**, Tech. Rep.UCB/ERL M92/41. UC Berkeley, Berkeley, 1992

SHANNON, Claude E.. A Symbolic Analysis of Relay and Switching Circuits. **Transactions of the American Institute of Electrical Engineers**, vol.57, no.12, pp.713,723, Dec. 1938.

SHANNON, Claude E.. The Synthesis of Two-Terminal Switching Circuits. **Bell System Technical Journal 28**, pp. 59–98, 1948.

SONG, N.; PERKOWSKI M.. A new approach to AND/OR/EXOR factorization for regular arrays, In: 24th Euromicro Conference, 1998, **Proceedings** of , vol.1, pp.269,276 vol.1, 25-27 Aug 1998

STANION, T.; SECHEN C.; A Method for Finding Good Ashenhurst Decompositions and its Application to FPGA Synthesis, 32nd Conference on Design Automation, **Proceedings** of, pp.60,64, 1995

SWORDS, S.; HUNT, W. A.; A Mechanically Verified AIG-to-BDD Conversion Algorithm, In: First international conference on Interactive Theorem Proving, **Proceedings** of, pp. 435-449, 2010.

TINDER, R. F.. Multilevel logic minimization using K-map XOR patterns, **IEEE Transactions on Education**, vol.38, no.4, pp.370-375, Nov 1995

TOGNI, J.D.; SCHNEIDER, F.R.; CORREIA, V.P.; RIBAS, R.P.; REIS, A.I.; Automatic generation of digital cell libraries, Symposium on Integrated Circuits and Systems Design, **Proceedings** of. 15th, pp.265-270, 2002

TURTON, B. C. H.. Extending Quine-McCluskey for Exclusive-Or logic synthesis, **IEEE Transactions on Education**, vol.39, no.1, pp.81-85, Feb 1996

VOLF, F. A. M.; JOZWIAK L.; L.; Decompositional logic synthesis approach for look up table FPGAs, IEEE International ASIC Conference and Exhibit, 1995, **Proceedings** of, pp.358-361, 18-22 Sep 1995.

WAGNER, Flávio R., André I. REIS, and Renato P. RIBAS. **Fundamentos de circuitos digitais**. Sagra Luzzatto, Porto Alegre, 2006

WANG, L. T.; CHANG, Y. W.; CHENG K. T.; **Electronic Design Automation: Synthesis, Verification and Test**. Morgan Kaufmann, 2009.

WEINBERGER; A. High-speed programmable logic array adders**; IBM Jorunal of Research & Development**, vol. 23, No. 2, pp. 163-178, March 1979.

WERBER, J.; RAUTENBACH, D.; SZEGEDY, C.; Timing optimization by restructuring long combinatorial paths. IEEE/ACM International Conference on Computer-Aided Design, **Proceedings** of, vol., no., pp.536,543, 4-8 Nov. 2007

WESTE, N.; HARRIS, D.. **Principles of CMOS VLSI Design - A Circuits and Systems Perspective,** 4th Edition Addison-Wesley, MA, 2010

YAMASHITA, S.; SAWADA, H.; NAGOYA, A. SPFD: a new method to express functional flexibility. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, vol. 19, no. 8. pp.840-49. Aug. 2000.

YANG, C.; CIESIELSKI, M.; BDS: A BDD-Based Logic Optimization System. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, vol. 21, no. 7, pp. 866-876, 2002.

YANG, S**.;** Logic Synthesis and Optimization Benchmarks User Guide Version 3.0. **Technical Report 1991-IWLS-UG-Saeyang**, MCNC Research Triangle Park, NC, January 1991.

YOSHIDA, H; FUJITA, M.. Exact minimum factoring of incompletely specified logic functions via quantified Boolean satisfiability, **IPSJ Transactions on System LSI Design Methodology**, vol. 4, pp.70-79, Feb. 2011.

YOSHIDA, H.; IKEDA, M.; ASADA, K.. Exact minimum logic factoring via quantified Boolean satisfiability**.** 13th IEEE International Conference on Electronics, Circuits and Systems, **Proceedings** of, pp.1065,1068, 10-13 Dec. 2006

ZHANG, R.; GUPTA, P.; JHA, N. K.. Synthesis of Majority and Minority Networks and Its Applications to QCA, TPL and SET Based Nanotechnologies, 18th International Conference on VLSI Design**, Proceedings** of, **pp. 229- 234**, **2005**

ZHANG, R.; WALUS, K.; WANG, W.; JULLIEN, G. A.. A method of majority logic reduction for quantum cellular automata, **IEEE Transactions on Nanotechnology**, vol.3, no.4, pp. 443- 450, 2004.