INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700  800/521-0600
Boolean and Multiple-Valued Functions in Combinational Logic Synthesis

by

Elena Vladimirovna Dubrova
Dipl. Eng., HIMEE, Sofia, Bulgaria, 1991

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming to the required standard

Dr. J. C. Muzio, Supervisor (Department of Computer Science)

Dr. J. A. Ellis, Departmental Member (Department of Computer Science)

Dr. D. M. Miller, Departmental Member (Department of Computer Science)

Dr. C. Morgan, Outside Member (Department of Philosophy)

Dr. I. G. Tabakow, External Examiner (HIMEE, Sofia, Bulgaria)

© Elena Vladimirovna Dubrova, 1997

University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.
Supervisor: Dr. J.C. Muzio

ABSTRACT

The subject of this dissertation is the theory of Boolean and multiple-valued functions. The main areas considered are: functional completeness, canonical forms, minimization of functions, discrete differences and functional decomposability. The results obtained are used as a foundation for the development of several new algorithms for logic synthesis of combinational logic circuits. These include an efficient algorithm for three-level AND-OR-XOR minimization for Boolean functions, an algorithm for generating the composition trees for Boolean and multiple-valued functions in a certain class, and an algorithm for computing a new canonical form of multiple-valued functions. Several other problems, related to logic synthesis, such as test generation for combinational logic circuits and synthesis of easily testable circuits are also addressed. Possible directions for future research are discussed.

Examiners:

Dr. J. C. Muzio, Supervisor (Department of Computer Science)

Dr. J. E. Ellis, Departmental Member (Department of Computer Science)

Dr. D. M. Miller, Departmental Member (Department of Computer Science)

Dr. C. Morgan, Outside Member (Department of Philosophy)

Dr. I. G. Tabakow, External Examiner (HIMEE, Sofia, Bulgaria)
## Contents

### Contents

iii

### List of Tables

vi

### List of Figures

vii

### Acknowledgment

ix

### Dedication

x

1  Introduction  1

2  Background  
   2.1  Notation  7  
   2.2  Basic notions  
      2.2.1  Relations  8  
      2.2.2  Functions  8  
      2.2.3  Binary operations  9  
   2.3  Chain-based Post algebra  9

3  A Canonical Form of MVL Functions  12
   3.1  Reed-Muller canonical form and its generalizations  13  
   3.2  The algebra  15  
   3.3  Functional completeness of \{\oplus, \cdot\}  17  
   3.4  Properties of the operations of the algebra B  19  
   3.5  Decomposition theorem  21  
   3.6  Canonical form of multiple-valued functions  23  
   3.7  An algorithm for constructing the canonical form  27  
   3.8  Conclusion  32

4  AND-OR-XOR Minimization of Boolean Functions  33
   4.1  Logic minimization  34  
   4.2  Notation and definitions  36  
   4.3  Upper bound on the number of product-terms in the AND-OR-XOR expansion  38  
   4.4  An algorithm for minimizing AND-OR-XOR expansions  40
# CONTENTS

4.4.1 Dividing the cubes into equivalence classes ........................................ 42
4.4.2 Obtaining $T_1$ and $T_2$ ................................................................. 45
4.4.3 Constructing $g_{1,init}$ and $g_{2,init}$ .................................................. 46
4.4.4 Determining common don't cares in $g_{1,init}$ and $g_{2,init}$ ............... 46
4.4.5 Multiple-output problems ............................................................... 48
4.5 Experimental Results ........................................................................ 48
4.6 Conclusion ....................................................................................... 52

5 Test Generation for Multiple-Valued Circuits ........................................ 53
5.1 Test generation for logic circuits ....................................................... 53
5.2 Boolean difference for test generation .............................................. 55
5.3 Definition of full sensitivity ............................................................... 56
5.4 Calculation of full sensitivity ............................................................. 57
5.5 Full sensitivity in test generation ....................................................... 62
  5.5.1 Test generation for primary inputs .............................................. 62
  5.5.2 Test generation for internal lines ............................................... 63
5.6 Total number of $m$-valued functions fully sensitive to all their variables 66
5.7 Conclusion ....................................................................................... 71

6 Composition Trees in Logic Synthesis .................................................. 73
6.1 Disjunctive decomposition of functions ............................................. 73
6.2 Composition trees ............................................................................ 77
6.3 An algorithm for constructing composition trees ............................... 81
6.4 Conclusion ....................................................................................... 87

7 Synthesis of Easily Testable Circuits .................................................... 89
7.1 Implementation of modulo $m$ sum-of-products canonical form ............ 90
7.2 Testability of internal lines ............................................................... 92
7.3 Testability of primary inputs ............................................................. 95
  7.3.1 A procedure for test generation ................................................. 96
  7.3.2 Evaluation of the effectiveness of the procedure ......................... 98
7.4 Testability by hardware redundancy ................................................ 101
7.5 Conclusion ....................................................................................... 102

8 Conclusion ....................................................................................... 104

Appendices

A Current-Mode CMOS Multiple-Valued Circuits .................................... 107
  A.1 Basic operations and symbols ......................................................... 108
    A.1.1 Constant-current source ......................................................... 108
    A.1.2 Current mirror ........................................................................ 109
    A.1.3 Current comparator ............................................................... 110
  A.2 Implementation of basic logic gates using current-mode CMOS circuits 111
    A.2.1 MIN and MAX circuits ......................................................... 113
    A.2.2 Addition modulo $m$ circuit ..................................................... 114
  A.3 Conclusion ..................................................................................... 120
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bibliography</td>
<td>122</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Benchmark results. ................................................................. 49
4.2 AOXMIN results for t4S1. ....................................................... 51
5.1 Truth table for the function from example. ......................... 59
A.1 HSPICE simulation parameters. ......................................... 112
List of Figures

3.1 Diagram illustrating the multiplication of $i$th row of the matrix $F^2$ by the matrix $T^2$. ........................................... 30

4.1 Karnaugh map of the function from the example. ......................... 37
4.2 Map of the example function. ................................................. 43
4.3 Implementation of the subroutine DivideEqClasses(). ................. 44
4.4 A function with all product-terms in the same equivalence class. ...... 46
4.5 Functions $g_{1,init}$ and $g_{2,init}$ for the function from the example. ... 47
4.6 Implementation of the subroutine SpecifyBoth(). ................. 47

5.1 Full sensitivities for the example function. ................................. 58
5.2 Example circuit. ........................................................................ 63
5.3 A multiple-valued logic circuit. .................................................... 64
5.4 Set $U$ with subsets $A_1, A_2, ..., A_n$. .................................. 66
5.5 Plots for $e(m,n)$ as a function of $n$ for fixed $m = 3.5$ and $10$. ... 68
5.6 Plots for $e(3,n)$ and $e^*(3,n)$ as functions of $n$ for $m = 3$. ... 70

6.1 Simple disjunctive decomposition ............................................. 74
6.2 Decomposition chart for an example function in $\Sigma$ .................. 75
6.3 Example of a composition tree ................................................. 78

7.1 Reed-Muller circuit. ..................................................................... 91
7.2 Circuit realizing modulo $m$ SOP form. ........................................ 91
7.3 Circuit implementing the function from the example. .................. 92
7.4 Circuit realizing modulo $m$ SOP form. ........................................ 94
7.5 Circuit with an extra multiplication mod $m$ gate $G^*$. ................ 101

A.1 Current source: (a) circuit configuration, (b) symbol. .................. 109
A.2 N-type and P-type current mirrors: (a),(c) circuit configurations, (b),(d) symbols. .................................................. 110
A.3 Classical current comparator. ....................................................... 110
A.4 Threshold current comparator of Onneweer and Kerkhoff. .......... 111
A.5 Minimum-maximum circuit of Onneweer and Kerkhoff. ............. 113
A.6 (a) Minimum and (b) Maximum circuits of Zhijian and Hong. ... 114
A.7 The simulation results for the MIN/MAX circuit of Onneweer and Kerkhoff: (a) $IA$, (b) $IB$, (c) MIN($IA,IB$), (d) MAX($IA,IB$), (e) power dissipation. ......... 115
A.8 The simulation results for the MIN and MAX circuits of Zhijian and Hong: (a) IA. (b) IB. (c) MIN(IA,IB). (d) MAX(IA,IB). (e) power dissipation. ........................................ 116
A.9 (a) Absolute difference circuit (b) Correction circuit. .................. 118
A.10 Addition modulo \( m \) circuit (a) circuit configurations. (b) symbolic scheme. ................................................................. 119
A.11 The simulation results for the addition modulo \( m \) circuit: (a) IA. (b) IB. (c) sum(IA,IB). (d) power dissipation. ......................... 121
Acknowledgment

I would like to thank Dr. Jon Muzio for his guidance and support throughout my graduate studies. Being a doctoral student and a new mother simultaneously wasn’t easy for me, and I warmly appreciate his patience and kindness.

I would also like to express my gratitude to Dr. Michael Miller for the many helpful discussions and for sharing with me his technical expertise, which inspired me in obtaining a number of results contained in this dissertation.

I am thankful to my committee members and my external examiner, Dr. Ivan Tabakow, for careful reviewing of this dissertation and their valuable comments and suggestions.

I am obliged to Dr. Michaela Serra and my fellow students in the VLSI group for attending my talks on multiple-valued logic and helping me to become a better lecturer. Ashraf Hafez, Bill Gardner, Claudio Costi and Kevin Cattell read and commented on the introduction of my dissertation. Stephen Goglin and Ken Kent suggested me the name of the tool described in Chapter 4. Enyu Wang helped me to prepare the slides for my defense. I greatly appreciate their kind help.

My special thanks to my husband Dr. Dilian Gurov for careful proofreading and numerous suggestions for improvements and corrections in the dissertation.

Finally, I would like to acknowledge the Natural Sciences and Engineering Research Council of Canada and the Canadian Microelectronic Corporation for providing research grants making my graduate studies possible.
DEDICATION

Dedication

To my father Vladimir Ivanovich Dubrova.
Chapter 1

Introduction

The main objects of study of this dissertation are discrete functions. While the theory of discrete functions is an interesting area of research on its own, it also has a direct practical application to logic synthesis. We study the properties of discrete functions and use them to develop several new algorithms for logic synthesis. Some problems related to logic synthesis, such as test generation for logic circuits and synthesis of easily testable circuits, are also addressed.

Discrete functions are mappings relating finite sets. In general, they may be heterogeneous, where the variables of the function do not take values in the same set. This dissertation, however, considers only the case of homogeneous functions of type $M^n \to M$ on a fixed set $M := \{0, 1, \ldots, m - 1\}$. This is a common restriction for logic synthesis-related work. Such functions are usually called multiple-valued or m-valued, and for the special case of $m = 2$, Boolean or switching functions.

Logic synthesis is a step in the design process for digital circuits. Generally, the design process depends heavily on the target technology. Integrated circuit technology progresses very quickly and the design methods used today might not be efficient in 10 years. However, logic synthesis is technology independent and therefore most of its techniques can be mapped into any underlying technology.

Logic synthesis starts with a description of a discrete function (by means of truth table, decision diagram, hardware description language) and produces a logic diagram
CHAPTER 1. INTRODUCTION

of the circuit implementing it. Such a diagram is usually called a logic circuit. A logic circuit has several inputs and one or more outputs, which take discrete values. It is composed of building blocks called logic gates from a selected set. The gates realize logic operations such as AND, OR and NOT, hence the name logic. Usually, the goal of logic synthesis is to find a minimal circuit realization of the function in terms of a given set of gates, under some criteria of minimality. The criteria might be reducing the number and size of gates that are needed to build the circuit, reducing the number of interconnections between these gates.

There are two types of logic circuits - combinational and sequential. In a combinational circuit, the output value depends only on the current value of the inputs. In a sequential circuit, the output depends on the current value of the inputs and on the past input values. A sequential circuit can be represented as combinational circuit with added memory devices or feedback loops. Therefore, a combinational circuit is a more fundamental building block. This dissertation deals only with combinational logic circuits, and we use the term "logic circuit" to mean "combinational logic circuit".

A discrete function models a combinational logic circuit by mapping the possible input assignments onto the values assumed by the output. The properties of discrete functions therefore provide a foundation for the methods of synthesis of combinational logic circuits.

If a logic circuit is composed of gates realizing Boolean functions, then such a circuit is called a Boolean or two-valued logic circuit. Likewise, a logic circuit built of gates realizing multiple-valued functions is called multiple-valued or \( m \)-valued. This dissertation, with the exception of Chapter 4, addresses the potential problems associated with multiple-valued logic circuits.

Our interest in multiple-valued logic circuits is twofold. First, we found that studying a problem in the general \( m \)-valued case gives us a better understanding of the underlying structure in the two-valued case because some properties, evident in
CHAPTER I. INTRODUCTION

the richer \( m \)-valued structure. often degenerate when restricted just to two values. Examples supporting this claim are shown in Chapter 5 and Chapter 7. Second, multiple-valued logic circuits offer several potential opportunities for the improvement of present very-large scale integrated (VLSI) circuit designs. Serious difficulties with limitations on the number of connections of an integrated circuit with the external world (pinout problem) as well as on the number of connections inside the circuit (interconnection problem) encountered in some VLSI circuit synthesis could be substantially reduced if signals in the circuit are allowed to assume four or more states rather than only two. If, for example, each connection carries twice as much information, then only half as many connections are required. Many laboratories world wide presently investigate possibilities for electronic fabrication of multiple-valued logic circuits. A recent achievement is the INTEL 16 Mbit flash memory chip with each cell of the memory capable of storing four discrete values \([67]\). Employing 4-valued logic allowed INTEL to drop the cost of the chip to \$20 per Mbyte. INTEL also declared that their longer term target is a 16-valued flash memory with a cost of 50 cents per Mbyte. We believe that the development of synthesis techniques for multiple-valued logic circuits is essential to facilitate their electronic fabrication. This motivated us in our research.

A fundamental role in logic synthesis is played by complete sets of functions. A set of functions is said to be functionally complete if any function can be defined as a composition of functions from this set. Theoretically, logic synthesis can be based upon any set of gates realizing a complete set of functions. In practice, however, the choice of gates to be used in logic synthesis is normally dictated by the cost of their implementation, which changes rapidly with progress in circuit technology. Other issues, influencing the choice of the basic set of gates, when realizing a given function, are:

- the existence of a simple expression for the function as a composition of functions from the basic set (implying the existence of an efficient circuit implementation
CHAPTER 1. INTRODUCTION

for the function), and

- the existence of a fast algorithm for computing this expression.

The search for a minimal expression for a given function (under certain criteria of minimality) without any restrictions on its structure, in terms of any practically meaningful complete set of functions, is known to be an extremely difficult task in terms of computational complexity. To be feasible, the practical algorithms for logic synthesis normally put some restrictions on the problem and seek for the solution to this restricted problem. Two common approaches are:

1. restrict the expression to be obtained to a particular type (e.g. two-level AND-OR expression)

2. restrict the functions for which the solution is sought to a particular class (e.g. symmetric functions, monotonic functions)

In this dissertation we study both approaches to logic synthesis. Chapter 3 and Chapter 4 follow approach (1) for two different types of restricted expressions. Chapter 6 follows approach (2) for a class of functions which is formally defined using a discrete difference introduced in Chapter 5. Chapter 5 also shows how this difference can be advantageously used for generating tests for multiple-valued logic circuits.

In certain cases, finding a minimal circuit realization for a given function is not the primary goal of logic synthesis. Such a situation may arise in specific applications where some other properties of the circuit are more important, like fault tolerance or safety. In Chapter 7 we develop a technique for logic synthesis, suitable for applications in which the ability to test circuits easily and quickly is critical.

The more detailed structure of the dissertation is as follows.

Chapter 2 describes the mathematical background for the dissertation.

In Chapter 3 we prove functional completeness of the set consisting of the operations of addition modulo $m$, minimum, and the set of all literal operators, where $m$ is
CHAPTER 1. INTRODUCTION

a positive integer. We show the existence of a canonical form for the multiple-valued functions in terms of these operations, and give the algorithm for the construction such a form. For the case \( m = 2 \), this canonical form reduces to a fixed polarity Reed-Muller canonical form, which is known to provide a basis for economical implementations of some practical Boolean functions [52]. We also show in the Appendix how the basic operators of the algebra can be implemented at the transistor level by CMOS current-mode technology.

In Chapter 4 we consider the realization of functions as the XOR of two AND-OR expressions, which is usually called AND-OR-XOR expansion. We develop an algorithm for minimizing AND-OR-XOR expansions. We also show that such an expansion has a smaller upper bound on the number of products than that of the AND-OR and AND-XOR expansions and, therefore, for some functions, results in simpler circuits.

Chapter 5 introduces a multiple-valued discrete difference, which we call full sensitivity, and show its application to generating tests for multiple-valued logic circuits. Full sensitivity is also used to define a class of functions \( \Sigma \), studied in Chapter 6. This class of functions was also independently considered by Bernhard von Stengel in [57]. He proved that all functions in this class have a unique representation, called a composition tree, which, if non-trivial, suggests the circuit realization of the function at a cost close to minimal. In Chapter 6, we present an efficient algorithm for generating such a representation.

In Chapter 7 we investigate the testability of circuits realizing modulo \( m \) sum-of-products forms. This canonical form has been extensively studied by many authors; however, its applications to logic synthesis have only been considered for the case \( m = 2 \). The circuits, realizing modulo 2 sum-of-products forms, are proved by Reddy [6] to be easily testable. We extend Reddy's result for \( m > 2 \). Generalizing from the two to the \( m \)-valued case, however, is shown to be a non-trivial problem, since for \( m > 2 \) several new phenomena occur which allow us to reduce the upper bound
on the number of tests required for fault detection, but make the generation of tests harder.

Chapter 8 summarizes the dissertation and suggests further work that could be undertaken from this research.
Chapter 2

Background

This chapter presents the necessary mathematical background for the dissertation. Most of its material is classical and is based on [2], [6], [14] and [44].

For convenience, this chapter includes all the general background. Background material that is specific to a single chapter is included with the chapter.

2.1 Notation

Throughout the dissertation we use "\(\cdot\)" for the minimum operation (also called MIN or, for the two-valued case, AND); "\(+\)" for the maximum operation (MAX or, for the two-valued case, OR); "\(\oplus\)" for the addition modulo \(m\) operation (XOR for the two-valued case); "\(\odot\)" for the multiplication modulo \(m\) operation; and "\(\not\)" for the complement operation (NOT). "\(\cdot\)" and "\(\odot\)" are omitted between adjacent variables, when this does not lead to any ambiguity.

We let \(M := \{0, 1, \ldots, m - 1\}\) be a finite set of values. We use early lower-case letters \(a, b, c, a_1, a_2, \ldots\) etc to denote elements over \(M\), and lower-case letters \(f, g, h, g_1, g_2, \ldots\) etc to denote functions. We use \(x_1, x_2, \ldots, x_n\) to denote variables of the functions, and use \(\mathcal{V} = \{1, 2, \ldots, n\}\) to denote the set of indices of these variables. We use capital letters \(A, B, C, \ldots\) etc for vectors or sets, and usually denote the elements of the set by indexed lower-case letters. For example, the elements of a set \(A\) are denoted as \(a_1, a_2, \ldots\). We use bold capital letters \(\mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots\) etc to denote matrices.
CHAPTER 2. BACKGROUND

2.2 Basic notions

This section describes briefly the fundamental notions of relation, function and operation.

2.2.1 Relations

Let \( A \) and \( B \) be sets. A binary relation \( R \) between \( A \) and \( B \) is a subset of the Cartesian product \( A \times B \). We use the notation \( aRb \) to denote that \( (a, b) \in R \).

Binary relations represent relationships between the elements of two sets. A more general type of relation is the \( n \)-ary relation, which expresses relationships among elements of more than two sets. However, this dissertation uses only binary relations, and therefore we do not introduce \( n \)-ary relations. In the following, we use the term "relation" to mean "binary relation".

Relations from a set \( A \) to itself are of special interest. A relation on \( A \) is a relation from \( A \) to \( A \), i.e. a subset of \( A \times A \).

Let \( R \) be a relation on \( A \) and let \( P \) be a property of binary relations (such as reflexivity, symmetry, or transitivity). The closure of \( R \) with respect to \( P \) is the least relation containing \( R \) that has \( P \).

A relation on a set \( A \) is called an equivalence relation if it is reflexive, symmetric, and transitive. Let \( R \) be an equivalence relation on \( A \). The set of all elements \( b \) of \( A \) such that \( bRa \) for an element \( a \in A \) is called the equivalence class of \( a \). The equivalence classes of \( R \) form a partition of \( A \).

2.2.2 Functions

A function \( f : A \rightarrow B \) from \( A \) to \( B \) is a relation which has the property that every element \( a \in A \) is the first element of exactly one ordered pair \( (a, b) \) of the relation.

So, a function \( f : A \rightarrow B \) assigns to each element \( a \in A \) a unique element \( b = f(a) \) in \( B \), called the image of \( a \). \( A \) is called the domain of \( f \) and \( B \) is called the codomain of \( f \). The range of \( f \) is the set of all images of elements of \( A \).
CHAPTER 2. BACKGROUND

A function \( f : A \to B \) can be specified by using a rule \( a \mapsto f(a) \). assigning to each element \( a \in A \) its image \( f(a) \) in \( B \).

A function \( f : A \to B \) is called injective when different elements of \( A \) always have different images or, in other word, if and only if \( a \neq b \) implies that \( f(a) \neq f(b) \).

A function \( f : A \to B \) is called surjective when the range is the whole codomain \( B \) or, in other words, if and only if for every element \( b \in B \) there is an element \( a \in A \) with \( f(a) = b \).

A function is called bijective when it is both injective and surjective.

In this dissertation we deal only with discrete functions of the type \( f : M^n \to M \) on a fixed set \( M := \{0, 1, \ldots, m - 1\} \), where \( M^n \) denotes the Cartesian product \( M \times M \times \ldots \times M \) of \( n \) sets \( M \). We say that \( f(x_1, \ldots, x_n) \) is an \( n \)-variable \( m \)-valued function. Such functions are called homogeneous, as opposed to heterogeneous functions, where the variables \( x_i \) of the function \( f(x_1, \ldots, x_n) \) do not take values in the same set. There are \( m^{(m^n)} \) homogeneous \( n \)-variable \( m \)-valued functions. For the special case of \( m = 2 \), \( m \)-valued functions are called switching or Boolean.

2.2.3 Binary operations

A binary operation \( \cdot \) on \( A \) is a function of type \( A \times A \to A \). So, a binary operation assigns to each ordered pair of elements \( (a, b) \) from \( A \times A \) a uniquely defined third element \( c = a \cdot b \) in the same set \( A \).

2.3 Chain-based Post algebra

In this section we describe a chain-based Post algebra, commonly used for representing multiple-valued functions. This algebra is a generalization of Shannon's switching algebra to the multiple-valued case.

Definition 2.1 A chain-based Post algebra is an algebra \( A = (M; J, +, \cdot : 0, m - 1) \).

where

\[(i)\ M := \{0, 1, \ldots, m - 1\} \text{ is the totally ordered carrier of } A;\]
(ii) $J := \{J_0, J_1, \ldots, J_{m-1}\}$ is a set of literal operators such that

$$J_i x := \begin{cases} m - 1 & \text{if } x = i \\ 0 & \text{otherwise} \end{cases}$$

where $x$ is a multiple-valued variable and $i \in M$ is a constant. For convenience, we write $J_i x$ as $x_i$:

(iii) $\cdot$ $+$ and $\cdot$ $-$ are the binary operations maximum (MAX) and minimum (MIN), respectively:

(iv) $0$ and $m - 1$ are constants of the algebra.

An algebra is functionally complete if it is based on a functionally complete set of operations. If constants need to be added to a set of operations to obtain a complete set, then such an algebra is called functionally complete with constants. The chain-based Post algebra is known to be functionally complete with constants [44].

The complement of a multiple-valued variable $x$ is defined in chain-based Post algebra as $x' := (m - 1) - x$, where $\cdot$ $-$ is the usual arithmetic subtraction.

Functional completeness of $A$ implies that every multiple-valued function can be expressed in terms of its operations. The next theorem shows a canonical form of any multiple-valued function in $A$. This form is said to be canonical because it gives a unique representation for multiple-valued functions. Throughout the dissertation, we refer to this form as the MIN-MAX canonical form. The sign $\sum$ used in the theorem stands for MAX.

**Theorem 2.2** Any $m$-variable function of $n$ variables has a unique expansion in $A$ of type

$$f(x_1, \ldots, x_n) = \sum_{i=0}^{m^n-1} c_i x_{i_1} x_{i_2} \cdots x_{i_n},$$

where $c_i \in M$ are constants, and $(i_1 i_2 \ldots i_n)$ is the $m$-ary expansion of $i$ with $i_1$ being the least significant digit.

In the two-valued case, the MIN-MAX canonical form reduces to the AND-OR canonical form. Notice, that in the two-valued case, $0 = x'$ and $1 = x$. An AND-OR
canonical form is often referred to as a "sum-of-product" form, but we prefer not to use this name to avoid confusion with the modulo $m$ sum-of-products form, cited in several chapters of the dissertation.

A function can be put into a MIN-MAX canonical form by a successive application of generalized Shannon decomposition to subfunctions of $f(x_1, \ldots, x_n)$. **Generalized Shannon decomposition** is an expansion of type:

$$ f(x) = \sum_{j \in M} \hat{f}_j f(x'_j) $$

where $x := (x_1, \ldots, x_n)$ and $x'_j$ is the vector $x$ with $x_i = j$, i.e.

$$ x'_j := (x_1, \ldots, x_{i-1}, j, x_{i+1}, \ldots, x_n). $$

with $j \in M, i \in \mathcal{V}$. So, in terms of these notations, $f(x'_j)$ denotes the subfunction of the function $f(x)$ with the variable $x_i$ being fixed to the value $j \in M$.

This concludes the background material for the dissertation.
Chapter 3

A Canonical Form of Multiple-Valued Functions

While complete sets of functions are widely studied for Boolean functions, less is known about the functionally complete sets for multiple-valued functions. In this chapter we show the functional completeness of the set consisting of the operations of addition modulo \( m \), minimum, and the set of all literal operators, where \( m \) is a positive integer. We prove the existence of a canonical form over this set, and give an algorithm for constructing this form. For the case \( m = 2 \), this canonical form reduces to a fixed polarity Reed-Muller canonical form, which is known to provide a suitable basis for the implementation of some practical Boolean functions [52].

The chapter is organized as follows. In Section 3.1, we define the Reed-Muller canonical form and give a summary of previous work on its generalization to the multiple-valued case. In Section 3.2, an algebra based on the operations of addition modulo \( m \), minimum, and the set of all literal operators is introduced. Section 3.3 presents a proof of the functional completeness (with constants) of the set consisting of addition modulo \( m \) and minimum operations. Section 3.4 describes the properties of the operations of the algebra needed in the proofs of the main results of the chapter. In Section 3.5, a decomposition, allowing a function of \( n \) variables to be expressed through \( n \) functions of \( n - 1 \) variables, is developed. Using this decomposition, in Section 3.6, a canonical form for the multiple-valued functions in terms of the oper-
CHAPTER 3. A CANONICAL FORM OF MVL FUNCTIONS

Ations of the algebra is derived. An algorithm for constructing the canonical form is presented in Section 3.7. Section 3.8 contains conclusion and suggestions for further research. In Appendix A, we show a CMOS transistor-level realization of the gates, implementing the basic operations of the algebra and simulation of these gates using the HSPICE program.

Some of the results in this chapter are contained in [22].

3.1 Reed-Muller canonical form and its generalizations

In 1954 Reed [49] and Muller [43] proved that any n-variable Boolean function has a canonical form in terms of AND and XOR operations of type:

\[ f(x_1, \ldots, x_n) = \sum_{i=0}^{2^n-1} c_i x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n}. \]  

(3.1)

where the sign \( \sum \) stands for XOR. \( c_i \in \{0, 1\} \) are constants, \((i_1i_2 \ldots i_n)\) is the binary expansion of \( i \) with \( i_1 \) being the least significant digit, and \( x_j^0 = 1 \) and \( x_j^1 = x_j \) for \( j \in N \). The form (3.1) is usually called Reed-Muller canonical form, after its inventors. All product-terms in (3.1) consist of uncomplemented variables only.

If the restriction that all the variables appear uncomplemented is removed, and variables are allowed to appear complemented as well, then the Reed-Muller canonical form extends to fixed polarity Reed-Muller canonical form, which is unique for a fixed polarity \( k \in \{0, 1, \ldots, 2^n - 1\} \) and is given by:

\[ f(x_1, \ldots, x_n) = \sum_{i=0}^{2^n-1} c_i k_1 x_1^{i_1} k_2 x_2^{i_2} \ldots k_n x_n^{i_n} \]  

(3.2)

where \( c_i \in \{0, 1\} \) are constants, \((i_1i_2 \ldots i_n)\) and \((k_1k_2 \ldots k_n)\) are the binary expansions of \( i \) and \( k \), respectively, with \( i_1 \) and \( k_1 \) being the least significant digits. The term \( k_j x_j^{i_j}, j \in N \) is defined as follows: \( 0 x_j^1 = x_j, 1 x_j^1 = x_j^1 \) and, for any \( k_j, k_j x_j^0 = 1 \). When \( k \) is fixed, this form is unique for a given function.
CHAPTER 3. A CANONICAL FORM OF MVL FUNCTIONS

The concept of a Reed-Muller canonical form can be extended to m-valued logic in several ways, depending on how the AND and XOR operations are generalized. The first generalization, based on the operations of addition and multiplication modulo \( m \), where \( m \) is a prime number, was proposed by Cohn in 1960 [9]. He proved that any function of \( n \) variables has a unique modulo \( m \) sum-of-products form of the type:

\[
f(x_1, \ldots, x_n) = \sum_{i=0}^{m^n-1} c_i \cdot x_{i_1}^{k_1} \cdot x_{i_2}^{k_2} \cdots x_{i_n}^{k_n},
\]

(3.3)

where the sign \( \sum \) stands for multiplication modulo \( m \), \( c_i \in M \) are constants, \((i_1i_2\ldots i_n)\) is the \( m \)-ary expansion of \( i \) with \( i_1 \) being the least significant digit, and the term \( x_{i_j}^{k_j} \) denotes the \( ij \)th power of the variable \( x_j \), \( j \in N \). Modulo \( m \) addition and multiplication form a Galois field of order \( m \).

Later this generalization was further extended by Pradhan [47] for the case when \( m \) is a power of a prime, i.e. \( m = p^k \) (\( p \) - a prime number, \( k \) - a positive integer).

Kodandapani and Setlur [34] proposed a generalization of (3.2), based on the operations of addition and multiplication modulo \( m \) (\( m \) - a prime number) and the set of all literal operators, which is unique for a fixed polarity \( k \in \{0,1,\ldots,m^n-1\} \). The form is of type:

\[
f(x_1, \ldots, x_n) = \sum_{i=0}^{m^n-1} c_i \cdot x_{i_1}^{k_1} \cdot x_{i_2}^{k_2} \cdots x_{i_n}^{k_n},
\]

(3.4)

where \( c_i \in M \) are constants, \((i_1i_2\ldots i_n)\) and \((k_1k_2\ldots k_n)\) are the \( m \)-ary expansions of \( i \) and \( k \), respectively, with \( i_1 \) and \( k_1 \) being the least significant digits, and the term \( i_j x_{j}^{k_j} \) equals \( m - 1 \) whenever \( i_j = 0 \), and equals \( i_j x_{j}^{k_j} \) otherwise.

Harking and Moraga [27] introduced an extension of Cohn’s form (3.3), where an additive transform \( x_j + k_j \) is performed on each variable \( x_j \), according to a fixed polarity \( k \in \{0,1,\ldots,m^n-1\} \). The form is of type:

\[
f(x_1 + k_1, x_2 + k_2, \ldots, x_n + k_n) = \sum_{i=0}^{m^n-1} c_i \cdot x_{i_1}^{i_1} \cdot x_{i_2}^{i_2} \cdots x_{i_n}^{i_n}
\]

(3.5)
where \( c_i \in M \) are constants. \((i_1i_2\ldots i_n)\) and \((k_1k_2\ldots k_n)\) are the \( m \)-ary expansions of \( i \) and \( k \), respectively, with \( i_1 \) and \( k_1 \) being the least significant digits. When \( k \) is fixed, this form is unique for a given function.

All of the above described generalizations are only applicable for the algebras with \( m \) being a prime or a power of a prime number. In this chapter we introduce a generalization of the fixed polarity Reed-Muller canonical form, based on the operations of addition modulo \( m \), minimum and the set of all literal operators, with \( m \) being any positive integer. An \( n \)-variable \( m \)-valued function has \( m^n \) such forms, each characterized by a fixed polarity \( k \in \{0, 1, \ldots, m^n - 1\} \) and a corresponding vector of coefficients \([c_0 c_1 \ldots c_{m^n-1}]\), \( c_j \in M \). The form is unique for a fixed \( k \). We present a procedure for computing the coefficients of such forms, based on matrix multiplication. The vectors of coefficients for different polarities are obtained simultaneously, which makes it possible to choose the canonical form with the minimal number of non-zero coefficients.

### 3.2 The algebra

The work in this chapter is based on a multiple-valued algebra \( B \) defined as follows:

**Definition 3.1** A multiple-valued algebra \( B \) is an algebra \( B = (M; \oplus, \leq, \cdot, J; 0, m - 1) \), where

(i) \( M := \{0, 1, \ldots, m - 1\} \) is the totally ordered carrier of \( B \):

(ii) \( \oplus \) is the binary operation addition modulo \( m \):

(iii) \( \cdot \) is the binary operation minimum (MIN):

(iv) \( J := \{J_0, J_1, \ldots, J_{m-1}\} \) is a set of literal operators:

(v) \( 0 \) and \( (m - 1) \) are constants of the algebra.

The operations \( \oplus \) and \( \cdot \) are commutative and associative. They do not distribute over each other. \( \cdot \) is idempotent. The constant \( 0 \) is the null element and
the constant \((m - 1)\) in the unit element of “\(\cdot\)”. The constant 0 is the unit element of “\(\oplus\)”. Recall that, for convenience, we write \(J_i x\) as \(\hat{x}\).

Every element \(a\) of \(M\) has an inverse \(-a\) (with respect to the “\(\oplus\)” operation). defined as

\[
-a := a \oplus a \oplus \cdots \oplus a. \quad \text{\(m-1\) times}
\]

In order to simplify the derivations below, we define the operations of complement and subtraction modulo \(m\). All operations are extended to functions as usual.

**Definition 3.2** The complement of a multiple-valued variable \(x\) is defined by

\[
x' := (m - 1) \oplus (-x)
\]

Obviously \(x \oplus x' = m - 1\) since for any \(x\) ranging in \(M\). \(x \oplus (-x) = 0\).

**Definition 3.3** Subtraction modulo \(m\) “\(\ominus\)” is defined by

\[
x \ominus y := x \ominus (-y)
\]

where \(x\) and \(y\) denote multiple-valued variables.

Using subtraction, the complement of an \(x\) can be represented as \(x' = (m - 1) \ominus x\).

The chain-based Post algebra (Definition 2.1), based on the operations \(\text{MIN}, \text{MAX}\) and the set of all literal operators, is well-known to be functionally complete with constants [44]. Since \(\text{MAX}\) can be expressed through \(\text{MIN}\) and complement using de Morgan’s law \(x + y = (x' \cdot y')', \) and since complement is defined through addition modulo \(m\) and the constant \((m - 1)\) (Definition 3.2), we can conclude that the algebra \(\mathcal{B}\) is also functionally complete with constants.

While the functional completeness of the set of operations \(\{\oplus, \cdot, J\}\) is quite obvious, a more interesting fact is that \(\mathcal{B}\) remains functionally complete at the suppression of literal operators \(J\) from the basic set, i.e. \(\mathcal{B}\) is complete (with constants) over the set \(\{\oplus, \cdot\}\). This is proved in the next section.
3.3 Functional completeness of \( \{ \oplus, \cdot \} \)

The following property shows that the literal operators can be expressed in terms of the operations \( \oplus \) and \( \cdot \) and the constant \( (m - 1) \), which proves the functional completeness (with constants) of \( \{ \oplus, \cdot \} \).

**Property 3.4** The literal \( \hat{x} \) can be expressed in terms of \( \oplus \), \( \cdot \) and \( ' \) as follows:

\[
\hat{x} = (g'(x, i) \oplus g(x, i) \cdot g'(x, i))'.
\]

where \( g(x, i) := (x \oplus i')(x' \oplus i) \).

**Proof:** 1) Let \( x = i \). Then \( \hat{x} = m - 1 \). On the other hand:

\[
g(i, i) = (i \oplus i')(i' \oplus i) \quad \{ \text{definition of } g(x, i) \}
\]

\[
= (m - 1) \cdot (m - 1) \quad \{ \forall a \in M : a \oplus a' = m - 1 \}
\]

\[
= (m - 1) \quad \{ \text{idempotency of } \cdot \}
\]

Therefore

\[
(g'(i, i) \oplus g(i, i) g'(i, i))' = ((m - 1)' \oplus (m - 1) \cdot (m - 1)')' \quad \{g(i, i) = m - 1\}
\]

\[
= (0 \oplus (m - 1) \cdot 0)' \quad \{ \text{Definition 3.2} \}
\]

\[
= (0 \oplus 0)' \quad \{0 \text{ is the null element of } \cdot \}
\]

\[
= 0' \quad \{0 \text{ is the unit element of } \oplus \}
\]

\[
= m - 1 \quad \{ \text{Definition 3.2} \}
\]

Hence, for \( x = i \) \( \hat{x} = (g'(x, i) \oplus g(x, i) \cdot g'(x, i))' \).

2) Let \( x \neq i \). Then \( \hat{x} = 0 \). On the other hand. we show below that (a) for each \( x \) and \( i \), \( x \neq i \) implies \( g(x, i) < \lfloor \frac{m}{2} \rfloor \), and further, (b) for any \( a < \lfloor \frac{m}{2} \rfloor \), it is true that \( (a' \oplus aa')' = 0 \).

a) We prove part (a) by showing that

\[
\forall x, i, x \neq i \left[ (x \oplus i' \geq \lfloor \frac{m}{2} \rfloor) \Rightarrow (x' \oplus i < \lfloor \frac{m}{2} \rfloor) \right].
\]
CHAPTER 3. A CANONICAL FORM OF MVL FUNCTIONS

1) \( x \oplus i' \geq \left\lfloor \frac{m}{2} \right\rfloor \) \{hypothesis\}

2) \( \left\lfloor \frac{m}{2} \right\rfloor \leq x \oplus i' < m - 1 \) \{\((x \neq i) \Rightarrow (x \oplus i' \neq m - 1)\)\}

3) \( \exists z [x \oplus i' \oplus z = m - 1] \) \{(\text{M.} \oplus) \text{ is a group}\}

4) \( 1 \leq z \leq \left\lfloor \frac{m}{2} \right\rfloor \) \{(2).\( \text{(3)} \)\}

5) \( (x \oplus x') \oplus (i \oplus i') = (m - 1) \oplus (m - 1) \) \{\( \forall a \in M : a \oplus a' = m - 1 \)\}

6) \( (x \oplus i') \oplus (x' \oplus i) \oplus z = (m - 1) \oplus (m - 1) \oplus z \) \{reordering\}

7) \( x' \oplus i = (m - 1) \oplus z \) \{(3).\( \text{(6)} \)\}

8) \( x' \oplus i = z \oplus 1 \) \{Definition 3.3. \(-1 = m - 1\)\}

9) \( x' \oplus i < \left\lfloor \frac{m}{2} \right\rfloor \) \{(4).\( \text{(8)} \)\}

Hence, for \( x \neq i \), \( x \oplus i' \geq \left\lfloor \frac{m}{2} \right\rfloor \) implies \( x' \oplus i < \left\lfloor \frac{m}{2} \right\rfloor \) and so \((x \oplus i')(x' \oplus i) < \left\lfloor \frac{m}{2} \right\rfloor \).

Consequently, \( g(x, i) < \left\lfloor \frac{m}{2} \right\rfloor \).

b) For any \( a < \left\lfloor \frac{m}{2} \right\rfloor \) it is true that:

\[
\begin{align*}
(a' \oplus aa')' &= (a' \oplus a)' \quad \{(a < \left\lfloor \frac{m}{2} \right\rfloor) \Rightarrow (aa' = a)\}
\ &= (m - 1)' \quad \{\forall a \in M : a \oplus a' = m - 1\}
\ &= 0 \quad \{\text{Definition 3.2}\}
\end{align*}
\]

Hence, for \( x \neq i \), \( x = (g'(x, i) \oplus g(x, i) \cdot g'(x, i))' \).

\[\square\]

Functional completeness of an algebra means that every multiple-valued function can be expressed in terms of its operations. In Section 3.6, we derive a canonical form, which gives a unique representation of any multiple-valued function in the algebra \( B \). Although our canonical form can be expressed in terms of \( \{\oplus, \cdot\} \) only, we use literal operators as well because this simplifies the form. It is easy to see that expanding the literal operators by applying Property 3.4 cannot result in further simplification of the form, since "\( \oplus \)" is not distributive over "\( \cdot \)".

The proof of existence of the canonical form is based on a number of properties establishing relationships between the operations of \( B \). These properties are presented and proved in the next section.
3.4 Properties of the operations of the algebra $B$

Let $f, g$ denote multiple-valued functions and $i, j, i \neq j,$ denote constants over $M$. The sign $\sum$ used in the properties and elsewhere throughout the chapter denotes addition modulo $m$.

**Property 3.5** The following properties hold:

- **a)** $(\hat{x})' = \sum_{j \in M - \{i\}} \hat{j}$
- **b)** $f \cdot \hat{x} = f \oplus [(-f) \cdot (\hat{x})']$
- **c)** $f \cdot \hat{x} + g \cdot \hat{x} = f \cdot \hat{x} \oplus g \cdot \hat{x}$
- **d)** $f \cdot (\hat{x} \oplus \hat{x}) = f \cdot \hat{x} \oplus f \cdot \hat{x}$
- **e)** $\hat{x} \cdot (f \oplus g) = \hat{x} \cdot f \oplus \hat{x} \cdot g$
- **f)** $\hat{x} \cdot (f \oplus g) = \hat{x} \cdot f \oplus \hat{x} \cdot g$

**Proof (a):**  
1) Let $x = i$. Then clearly $\sum_{j \in M - \{i\}} \hat{j} = 0$. On the other hand, $(\hat{x})' = (m - 1)' = 0$.

2) Let $x \neq i$. Then there exists exactly one $k$ in $M$ such that $x = k$ and so $\hat{k} = m - 1$. Consequently $\sum_{j \in M - \{i\}} \hat{j} = m - 1$. On the other hand $(\hat{x})' = 0' = m - 1$.

Hence for both cases $(\hat{x})' = \sum_{j \in M - \{i\}} \hat{j}$.

**Proof (b):**  
1) Let $x = i$. Then $f \cdot \hat{i} = f$. On the other hand:

$$f \oplus [(-f) \cdot (\hat{x})'] = f \oplus [(-f) \cdot (m - 1)']$$

$$= f \oplus [(-f) \cdot 0]$$

$$= f \oplus 0$$

$$= f$$

2) Let $x \neq i$. Then $f \cdot \hat{i} = 0$. On the other hand:
CHAPTER 3. A CANONICAL FORM OF MVL FUNCTIONS

\[
f \oplus [(-f) \cdot (\dot{x})'] = f \oplus [(-f) \cdot 0'] \\
= f \oplus [(-f) \cdot (m - 1)] \\
= f \oplus f \\
= 0
\]

Hence for both cases \( f \cdot \dot{x} = f \oplus [(-f) \cdot (\dot{x})'] \)

(c): Since \( i \neq j \), \( x \) cannot be equal to both \( i \) and \( j \) at once. It is always the case that either \( x \neq i \) or \( x \neq j \) or both. Let \( x \neq i \). Then the left hand side is \( f \cdot \dot{x} + g \cdot \dot{x} = f \cdot 0 + g \cdot \dot{x} = 0 + g \cdot \dot{x} = g \cdot \dot{x} \), and the right hand side is \( f \cdot \dot{x} \oplus g \cdot \dot{x} = f \cdot 0 \oplus g \cdot \dot{x} = 0 \oplus g \cdot \dot{x} = g \cdot \dot{x} \).

Hence for \( x \neq i \), \( f \cdot \dot{x} + g \cdot \dot{x} = f \cdot \dot{x} \oplus g \cdot \dot{x} \). For the other cases the proof is similar.

(d): Since \( i \neq j \), \( x \) cannot be equal to both \( i \) and \( j \) at once. It is always the case that either \( x \neq i \) or \( x \neq j \) or both. Let \( x \neq i \). Then the left hand side is \( f(\dot{x} \oplus \dot{x}) = f \cdot (0 \oplus \dot{x}) = f \cdot \dot{x} \), and the right hand side is \( f \cdot \dot{x} \oplus f \cdot \dot{x} = f \cdot 0 \oplus f \cdot \dot{x} = f \cdot \dot{x} \).

Hence for \( x \neq i \), \( f \cdot (\dot{x} \oplus \dot{x}) = f \cdot \dot{x} \oplus f \cdot \dot{x} \). For the other cases the proof is similar.

(e): 1) Let \( x = i \). Then the left hand side is \( \dot{x} \cdot (f \oplus g) = (m - 1) \cdot (f \oplus g) = f \oplus g \), and the right hand side is \( \dot{x} \cdot f \oplus \dot{x} \cdot g = (m - 1) \cdot f \oplus (m - 1) \cdot g = f \oplus g \).

2) Let \( x \neq i \). Then the left hand side is \( \dot{x} \cdot (f \oplus g) = 0 \cdot (f \oplus g) = 0 \), and the right hand side is \( \dot{x} \cdot f \oplus \dot{x} \cdot g = 0 \cdot f \oplus 0 \cdot g = 0 \).

Hence for both cases \( \dot{x} \cdot (f \oplus g) = \dot{x} \cdot f \oplus \dot{x} \cdot g \).

(f): 1) Let \( x = i \). Then the left hand side is \( \dot{x} \cdot (f \oplus g) = (m - 1) \cdot (f \oplus g) = f \oplus g \), and the right hand side is \( \dot{x} \cdot f \oplus \dot{x} \cdot g = (m - 1) \cdot f \oplus (m - 1) \cdot g = f \oplus g \).

2) Let \( x \neq i \). Then the left hand side is \( \dot{x} \cdot (g \oplus f) = 0 \cdot (f \oplus g) = 0 \), and the right hand side is \( \dot{x} \cdot f \oplus \dot{x} \cdot g = 0 \cdot f \oplus 0 \cdot g = 0 \).
Hence for both cases \( \hat{x} \cdot (f \oplus g) = \hat{x} \cdot f \oplus \hat{x} \cdot g. \)

\[ \square \]

### 3.5 Decomposition theorem

In this section we present a decomposition allowing a function of \( n \) variables to be expressed through \( m \) functions of \( n - 1 \) variables. This decomposition can be considered as a generalization of the positive and negative decompositions of Boolean functions to the multiple-valued case. Recall from Chapter 2 that \( f(x_i') \) denotes a subfunction of the function \( f(x) \) with the variable \( x_i \) being fixed to the value \( j \), i.e. \( f(x_i') = f(x_1, \ldots, x_{i-1}, j, x_{i+1}, \ldots, x_n) \). Then, the positive and negative decompositions of Boolean functions are of form [36]:

\[
\begin{align*}
  f(x) &= f(x^0) \uplus x_n(f(x^0) \uplus f(x^1)) & \text{positive decomposition} \\
  &= f(x^1) \uplus x_n(f(x^0) \uplus f(x^1)) & \text{negative decomposition}
\end{align*}
\]

Theorem 3.6 is the general decomposition theorem for a function \( f(x) \) about some variable \( x_i \). However, for notational convenience, the theorem is stated and proved for decompositions about the least significant variable \( x_n \).

**Theorem 3.6 (Decomposition Theorem)** Every \( m \)-valued function \( f(x) \) can be decomposed with respect to the variable \( x_n \) and a given \( i \in M \) in the following way:

\[
f(x) = f(x_n') \uplus \sum_{j=1}^{m-1} (f(x_n^{i \oplus j}) \uplus f(x_n^i)) x_n^{i \oplus j}
\]

**Proof:**

Using generalized Shannon decomposition (2.1) we can express the function \( f(x) \) as follows:
In the above derivation we expanded \( \tilde{x}_n \cdot f(x^0_n) \) using Property 3.5(b). If alternatively we expanded \( \tilde{x}_n \cdot f(x^i_n) \) for some \( i \neq 0 \), then the derivation gives the proof for the corresponding value of \( i \).

\[ \square \]

For example, a 3-valued 2-variable function \( f(x_1, x_2) \) can be decomposed with respect to the variable \( x_2 \) and a given \( i \in \{0, 1, 2\} \) in the following way:

\[
f(x_1, x_2) = f(x^i_1) \circ [f(x^i_2) \circ f(x^0_2)] \circ [(f(x^i_2) \circ f(x^0_2)) \circ f(x^0_2)]
\]

The decompositions for all three possible values of \( i \) are:

For \( i = 0 \):
\[
f(x_1, x_2) = f(x^0_2) \circ [f(x^0_2) \circ f(x^0_2)] \circ [(f(x^0_2) \circ f(x^0_2)) \circ f(x^0_2)]
\]

For \( i = 1 \):
\[
f(x_1, x_2) = f(x^1_2) \circ [f(x^1_2) \circ f(x^1_2)] \circ [(f(x^1_2) \circ f(x^1_2)) \circ f(x^1_2)]
\]

For \( i = 2 \):
\[
f(x_1, x_2) = f(x^2_2) \circ [f(x^2_2) \circ f(x^2_2)] \circ [(f(x^2_2) \circ f(x^2_2)) \circ f(x^2_2)]
\]

For example, suppose \( f(x_1, x_2) \) is defined by the table below:
CHAPTER 3. A CANONICAL FORM OF MVL FUNCTIONS

Then we have

<table>
<thead>
<tr>
<th>( x_2 )</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

and, for the decomposition with respect to \( x_2 \) with \( i = 0 \)

\[
f(x_1, x_2) = f(x_2^0) \oplus \left[(f(x_2^0) \oplus f(x_2^1)) \cdot \frac{1}{x_2}\right] \oplus \left[(f(x_2^2) \oplus f(x_2^0)) \cdot \frac{1}{x_2}\right]
\]

where \( g_1(x_1) \) and \( g_1(x_1) \) are functions defined as follows:

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Obviously, if each of the subfunctions \( f(x_j^k) \), \( j \in M \), in the decomposition of \( f(x) \)
is successively decomposed about the remaining variables, we finally get an expression in which \( f(x) \) is expanded in all its variables. Since for each subfunction the decomposition can be made with respect to some constant \( i \in M \), there are \( m^n \) different ways to expand the function \( f(x) \) in all \( n \) variables. In the next section we prove that, for a fixed \( i \), each of these \( m^n \) expansions is a canonical form uniquely representing a multiple-valued function and show how to find these expansions directly, i.e. without applying step-by-step decomposition.

3.6 Canonical form of multiple-valued functions

In a two-valued system, any Boolean function of \( n \) variables has \( 2^n \) fixed polarity Reed-Muller canonical forms of type (3.2). In such an expansion, each variable \( x_i \) is
either in a complemented, or in an uncomplemented form, according to some polarity vector \( k = (k_n \ldots k_2 k_1) \). If \( k_i = 1 \) the variable \( x_i \) appears in complemented form, otherwise \( x_i \) appears in uncomplemented form. For example, the polarity vector \( k = (011) \) implies that \( x_1 \) and \( x_2 \) appear complemented in the Reed-Muller canonical form, and \( x_3 \) appears uncomplemented (\( x_1 \) is the lowest order variable). A polarity can be given not only as a binary vector \( (k_n \ldots k_2 k_1) \), but also as a decimal number \( k \in \{0.1, \ldots, 2^n - 1\} \), whose binary expansion is this binary vector. For example, for \( k = (011) \) the polarity can be given as \( k = 3 \).

We generalize the notion of fixed polarity for multiple-valued logic, assuming that in a fixed polarity form each variable \( x_i, i \in \{1, \ldots, n\} \), is represented by all literals except \( \bar{x}_i \), where \( (k_n \ldots k_2 k_1) \) is the \( m \)-ary expansion of a polarity \( k \in \{0.1, \ldots, m^n - 1\} \), given as a decimal number. For example, if \( m = 3 \), polarity vector \( k = (021) \) implies that \( x_1 \) is represented by literals \( \bar{x}_1 \) and \( \bar{x}_1 \) in the canonical form, \( x_2 \) by literals \( \bar{x}_2 \) and \( \bar{x}_2 \), and \( x_3 \) by literals \( \bar{x}_3 \) and \( \bar{x}_3 \). Similar generalization of polarity was used in [34].

The following theorem shows that there exist \( m^n \) canonical forms of a \( m \)-valued \( n \)-variable function, each characterized by a polarity \( k \in \{0.1, \ldots, m^n - 1\} \) and a corresponding vector of coefficients \( [c_0 c_1 \ldots c_{m^n - 1}] \). The notation \( ^i x^j \) used in the theorem below is defined as follows:

\[
^i x^j := \begin{cases} 
m - 1 & \text{if } i = 0 \\ 
^i \bar{x}^j & \text{otherwise}
\end{cases}
\]

where \( x \) is a multiple-valued variable and \( i, j \in M \) are constants. We denote by \( f_j, \)
\( j \in \{0, \ldots, m^n - 1\} \), the coefficients from the truth table for \( f(x_1, \ldots, x_n) \), with \( x_1 \) the lowest order variable.
Theorem 3.7 Any m-valued n-variable function can be expressed in a canonical form with a fixed polarity \( k \in \{0, 1, \ldots, m^n - 1\} \) as:

\[
f(x_1, \ldots, x_n) = \sum_{j=0}^{m^n-1} c_j x_1^{j_1} x_2^{j_2} \cdots x_n^{j_n}
\]

where \( c_j \in M \) are constants, and \( (j_1 \ldots j_n) \) and \( (k_n \ldots k_2 k_1) \) are the m-ary expansions of \( j \) and \( k \), respectively, with \( j_1 \) and \( k_1 \) being the least significant digits.

Proof: By induction on \( n \).

1) Let \( n = 1 \). According to Theorem 3.6, any function of one variable \( x \) can be decomposed with respect to this variable and a given \( i \in M \) as:

\[
f(x) = f_i = \sum_{j=0}^{m-1} (f_{i-j} \div f_i)^{i-j} x^j \quad \{\text{Theorem 3.6}\}
\]

\[
= c_0 \div \sum_{j=0}^{m-1} c_j x^j \quad \{\text{where } c_0 = f_i \text{ and } c_j = f_{i-j} \div f_i}\}
\]

\[
= c_0 \div x^0 \sum_{j=0}^{m-1} c_j x^j \quad \{0 x^i = m - 1 \text{ and } j x^i = x^j \text{ for } j \neq 0\}
\]

\[
= \sum_{j=0}^{m-1} c_j x^j
\]

which is the canonical form for \( n = 1 \) and polarity \( i \in \{0, 1, \ldots, m - 1\} \).

2) Hypothesis: Assume the result for all functions of \( n \) variables. According to Theorem 3.6, any function of \( n + 1 \) variables can be decomposed with respect to the variable \( x_{n+1} \) and a given \( i \in M \) as:

\[
f(x_1, \ldots, x_{n+1}) = f(x_{n+1}) \div \sum_{p=1}^{m-1} (f(x_{n+1}^{i-p}) \div f(x_{n+1}^i)) x_{n+1}^p
\]

By the induction hypothesis, which assumes the result for the functions of \( n \) variables, we can express each of the subfunctions \( f(x_{n+1}^i), f(x_{n+1}^{i-p}) \), \( p \in M \) in the canonical form for some polarity \( k = (k_n \ldots k_2 k_1) \). We use the notation \( c_j^c \) to denote the \( j \)th coefficient of the canonical form of the subfunction \( f(x_{n+1}^i) \). Then we have:
\[ f(x_1, \ldots, x_{n+1}) = \sum_{j=0}^{m-1} c_j^i j x_1^{k_1} \ldots j_n x_n^{k_n} \]
\[ \equiv \sum_{p=1}^{m-1} \left( \sum_{j=0}^{m-1} c_j^{i \ominus p} j x_1^{k_1} \ldots j_n x_n^{k_n} \right) x_{n+1}^{i \ominus p} \]

To simplify the exposition, we use the notation \( jX^k \) to stand for the term \( j x_1^{k_1} \ldots j_n x_n^{k_n} \).

Then the above expression becomes:

\[ f(x_1, \ldots, x_{n+1}) = \sum_{j=0}^{m-1} c_j^i jX^k \equiv \sum_{p=1}^{m-1} \left( \sum_{j=0}^{m-1} c_j^{i \ominus p} jX^k \equiv \sum_{j=0}^{m-1} c_j^i jX^k \right) x_{n+1}^{i \ominus p} \]

\[ \equiv \sum_{p=1}^{m-1} \left( \sum_{j=0}^{m-1} c_j^{i \ominus p} jX^k \equiv \sum_{j=0}^{m-1} c_j^i jX^k \right) x_{n+1}^{i \ominus p} \]

\[ = \sum_{j=0}^{m-1} c_j^i jX^k \equiv \sum_{p=1}^{m-1} \left( \sum_{j=0}^{m-1} c_j^{i \ominus p} jX^k \equiv \sum_{j=0}^{m-1} c_j^i jX^k \right) x_{n+1}^{i \ominus p} \]

\[ \{ \text{Definition 3.3} \} \]

\[ = \sum_{j=0}^{m-1} c_j^i jX^k \equiv \sum_{p=1}^{m-1} \left( - \sum_{j=0}^{m-1} c_j^i jX^k \right) x_{n+1}^{i \ominus p} \]

\[ \{ \text{distributivity of } \ominus \text{ over } \ominus \} \]

\[ = \sum_{j=0}^{m-1} c_j^i jX^k \equiv \sum_{p=1}^{m-1} \left( \sum_{j=0}^{m-1} c_j^{i \ominus p} jX^k \equiv \sum_{j=0}^{m-1} c_j^i jX^k \right) x_{n+1}^{i \ominus p} \]

\[ \{ \text{commutativity of } \ominus \ominus \} \]

\[ = \sum_{j=0}^{m-1} c_j^i jX^k \equiv \sum_{p=1}^{m-1} \left( \sum_{j=0}^{m-1} c_j^{i \ominus p} jX^k \equiv c_j^i jX^k \right) x_{n+1}^{i \ominus p} \]

\[ \{ \text{Definition 3.3} \} \]

\[ = \sum_{j=0}^{m-1} c_j^i jX^k \equiv \sum_{p=1}^{m-1} \left( \sum_{j=0}^{m-1} c_j^{i \ominus p} \equiv c_j^i \right) jX^k x_{n+1}^{i \ominus p} \]

\[ \{ \text{Property 3.5(f)} \} \]

\[ = \sum_{j=0}^{m-1} c_j^i jX^k \equiv \sum_{p=1}^{m-1} \sum_{j=0}^{m-1} \left( c_j^{i \ominus p} \equiv c_j^i \right) jX^k x_{n+1}^{i \ominus p} \]

\[ \{ \text{Property 3.5(f)} \} \]

\[ = \sum_{j=0}^{m-1} c_j^i jX^k 0 x_{n+1}^{i} \equiv \sum_{p=1}^{m-1} \sum_{j=0}^{m-1} \left( c_j^{i \ominus p} \equiv c_j^i \right) jX^k p x_{n+1}^{i} \]

\[ \{ 0 x^i = m - 1, \text{ and } p x^i = \bar{x}, \text{ for } p \neq 0 \} \]
CHAPTER 3. A CANONICAL FORM OF MVL FUNCTIONS

\[
= \sum_{j=0}^{m^n-1} c_j \sum_{j=0}^{m^n-1} (c_j^{i<j} \oplus c_j^{i=}) \sum_{j=0}^{m^n-1} (c_j^{i>(m-1)} \oplus c_j^{i=})
\]

\[
= \sum_{j=0}^{m^{n+1}-1} c_j \sum_{j=0}^{m^n-1} (c_j^{i<j} \oplus c_j^{i=}) \sum_{j=0}^{m^n-1} (c_j^{i>(m-1)} \oplus c_j^{i=})
\]

where \(c_j = c_j^{i=}\) for \(0 \leq j \leq m^n - 1\) and \(c_{pm^n+j} = (c_j^{i<p} \oplus c_j^{i=})\) for \(1 \leq p \leq m-1\). \(0 \leq j \leq m^n - 1\).

This is the canonical form for a function of \(n + 1\) variables for polarity vector \(k = (k_{n+1}k_n \ldots k_2k_1)\), where \(k_{n+1} = i\). This proves the theorem.

\[\Box\]

For example, the canonical form of a 3-valued 2-variable function \(f(x_1,x_2)\) for every given fixed polarity \(k\), \(k \in \{0,1,\ldots,8\}\) is given by:

\[
f(x_1,x_2) = \sum_{j=0}^{8} c_j x_1^{k_1} x_2^{k_2} = c_0 \oplus c_1 x_1^{1 \oplus k_1} \oplus c_2 x_1^{2 \oplus k_1} \oplus c_3 x_1^{3 \oplus k_1} \oplus c_4 x_2^{1 \oplus k_2} \oplus c_5 x_2^{2 \oplus k_2} \oplus c_6 x_2^{3 \oplus k_2} \oplus c_7 x_2^{4 \oplus k_2} \oplus c_8 x_2^{5 \oplus k_2} \oplus \]

where \((k_2k_1)\) is the ternary expansion of \(k\). For example, for \(k = 1\) the ternary expansion is \((k_2k_1) = (01)\) and the function has the following form:

\[
f(x_1,x_2) = c_0 \oplus c_1 x_1^{2} \oplus c_2 x_1^{0} \oplus c_3 x_2^{1} \oplus c_4 x_1^{1} x_2^{0} \oplus c_5 x_1^{0} x_2^{1} \oplus c_6 x_2^{2} \oplus c_7 x_1^{2} x_2^{0} \oplus c_8 x_1^{1} x_2^{0} \).
\]

In the next section we present an algorithm for computing the coefficients of the new canonical form.

3.7 An algorithm for constructing the canonical form

There are two main approaches to computing the coefficients of a fixed polarity form. One is to define a set of transformation matrices, one for each value of the polarity.
and to generate the coefficients for polarity \( k \) by multiplying the truth vector of the function by the \( k \)th transformation matrix. In our case, the definition of the transformation matrices follow from Theorem 3.6, and each of the matrices is, in fact, a permutation of the basic transformation matrix for zero polarity.

The second approach is to have a single transformation matrix, but to permute the values of the truth vector of the function to obtain the coefficients for different polarities. The way the truth vector values should be permuted is again suggested by Theorem 3.6. This approach seems to us more convenient and we make it the basis of our algorithm. We group truth vectors of the function, suitably permuted for each polarity, in a matrix \( F^n \), defined next.

Let \( F_u^p \) denotes a matrix whose rows correspond to certain permutations of the values of the truth vector of the subfunction \( f_u(x_1, \ldots, x_p) = f(x_1, \ldots, x_p, u_1, \ldots, u_{n-p}) \) of \( f(x_1, \ldots, x_n) \), where \((u_1, \ldots, u_{n-p})\) is the \( m \)-ary expansion of \( u \). \( u \in \{0, \ldots, m^{n-p} - 1\} \), with \( u_1 \) being the least significant digit, i.e. \( u = \sum_{q=1}^{n-p} m^{q-1} u_q \). and \( p \in \{1, 2, \ldots, n-1\} \). We denote by \( f_u \) for the subfunctions of zero variables, which are the coefficients from the truth table for \( f(x_1, \ldots, x_n) \). The notation \([F^p_u]_{ij}\) used in the definition refers to the submatrix in the \( i \)th row and \( j \)th column of the matrix \( F_u^p \).

**Definition 3.8** The \( m^n \times m^n \) matrix \( F^n \) is defined as \( F_0^u \), where \( F_u^p \) is defined inductively by:

1. \( F_0^0 := [f_u] \)
2. \([F_u^p]_{ij} := F_{um+i(v_{ij})}^{p-1} \) with \( i, j \in M \), \( p \in N \), \( u \in \{0, \ldots, m^{n-p} - 1\} \).

The scheme is obvious from an example. For instance, for \( m = 3 \), \( n = 2 \) the matrix \( F^2 \) is constructed as follows:

1. For \( u \in \{0, 1, 2\} \), \( F_0^u := [f_u] \).
2. \( F_0^1 = \begin{bmatrix} f_0 & f_1 & f_2 \\ f_1 & f_2 & f_0 \\ f_2 & f_0 & f_1 \end{bmatrix} \), \( F_1^1 = \begin{bmatrix} f_3 & f_4 & f_5 \\ f_4 & f_5 & f_3 \\ f_5 & f_3 & f_4 \end{bmatrix} \), \( F_1^2 = \begin{bmatrix} f_6 & f_7 & f_8 \\ f_7 & f_8 & f_6 \\ f_8 & f_6 & f_7 \end{bmatrix} \).
CHAPTER 3. A CANONICAL FORM OF MVL FUNCTIONS

So:

$$\mathbf{F}^2 = \mathbf{F}_0^2 = \begin{bmatrix} \mathbf{F}_0^1 & \mathbf{F}_1^1 & \mathbf{F}_2^1 \\ \mathbf{F}_0^1 & \mathbf{F}_1^1 & \mathbf{F}_0^1 \\ \mathbf{F}_2^1 & \mathbf{F}_0^1 & \mathbf{F}_1^1 \end{bmatrix} = \begin{bmatrix} f_0 & f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 \\ f_1 & f_2 & f_0 & f_4 & f_5 & f_3 & f_7 & f_8 & f_6 \\ f_2 & f_0 & f_1 & f_5 & f_3 & f_4 & f_8 & f_6 & f_7 \\ f_3 & f_4 & f_5 & f_6 & f_7 & f_8 & f_0 & f_1 & f_2 \\ f_4 & f_5 & f_3 & f_7 & f_8 & f_6 & f_1 & f_2 & f_0 \\ f_5 & f_3 & f_4 & f_8 & f_6 & f_7 & f_2 & f_0 & f_1 \\ f_6 & f_7 & f_8 & f_0 & f_1 & f_2 & f_3 & f_4 & f_5 \\ f_7 & f_8 & f_6 & f_1 & f_2 & f_0 & f_4 & f_5 & f_3 \\ f_8 & f_6 & f_7 & f_2 & f_0 & f_1 & f_5 & f_3 & f_4 \end{bmatrix}.$$  

The definition below presents a transformation matrix $\mathbf{T}^n$, needed to obtain the coefficients of the canonical form from the matrix $\mathbf{F}^n$.

**Definition 3.9** The $m^n \times m^n$ matrix $\mathbf{T}^n$ is defined inductively by:

1. $\mathbf{T}^0 = [1]$  
2. $\mathbf{T}^n := \begin{bmatrix} \mathbf{T}^{n-1} & (m-1) \mathbf{T}^{n-1} & (m-1) \mathbf{T}^{n-1} & \ldots & (m-1) \mathbf{T}^{n-1} \\ 0 & \mathbf{T}^{n-1} & 0 & \ldots & 0 \\ 0 & 0 & \mathbf{T}^{n-1} & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & \mathbf{T}^{n-1} \end{bmatrix}$  

where $\mathbf{\cdot}$ denotes multiplication modulo $m$.

For example, for $m = 3$ the corresponding matrices $\mathbf{T}^0$, $\mathbf{T}^1$ and $\mathbf{T}^2$ are as follows:

$$\mathbf{T}^0 = [1], \quad \mathbf{T}^1 = \begin{bmatrix} 122 \\ 010 \\ 001 \end{bmatrix}, \quad \mathbf{T}^2 = \begin{bmatrix} 122 & 211 & 211 \\ 010 & 020 & 020 \\ 001 & 002 & 002 \end{bmatrix}.$$  

To find the coefficients of a fixed polarity canonical form of a multiple-valued function $f(x)$, the matrix $\mathbf{F}^n$ is multiplied by the transformation matrix $\mathbf{T}^n$:

$$\mathbf{C} = \mathbf{F}^n \mathbf{T}^n$$
CHAPTER 3. A CANONICAL FORM OF MVL FUNCTIONS

Every row \( k \) of the resulting \( m^n \times m^n \) matrix \( C \) corresponds to the vector of coefficients \([c_0 \ c_1 \ldots \ c_{m^n-1}]\) with polarity \( k \). The proof of this result follows directly from Theorem 3.6 and Theorem 3.7.

The step-by-step execution of the operation \( F^n \circ T^n \) involves the summation of a total \( m^n \times m^n \times m^n \) individual product terms (\( \times \) here denotes a regular arithmetic multiplication). However, due to the regular structure of the matrix \( T^n \) a "fast" procedure for performing \( F^n \circ T^n \) is possible, with the total number of summations reduced to \( n \times \frac{m^n-1}{m} \times m^n \times m^n \). which makes the complexity of the procedure \( O(N^2 \lg N) \), where \( N = m^n \). A fragment of the graphical representation of the fast procedure for \( n = 2, m = 3 \) is shown on Figure 3.1. The diagram shown is similar to a butterfly diagram for the Fast Fourier Transform [29]. The diagram illustrates the multiplication of the \( i \)th row of \( F^2 \) by \( T^2 \). The result is a vector of coefficients \([c_0 c_1 \ldots c_8]\) with polarity \( i \). The total number of summations required is \( n \times \frac{m^n-1}{m} \times m^n = 12 \). The complete diagram for multiplication of \( F^2 \) by \( T^2 \) consists of \( m^n = 9 \) such fragments.

![Diagram](image)

Figure 3.1: Diagram illustrating the multiplication of \( i \)th row of the matrix \( F^2 \) by the matrix \( T^2 \).

The following example illustrates the calculation of the matrix of coefficients \( C \).
Example 3.10. Consider the following 3-valued 2-variable function:

\[
f(x_1, x_2) = 1 \cdot x_1 \cdot x_2 + 1 \cdot \bar{x}_1 \cdot x_2 + 2 \cdot x_1 \cdot \bar{x}_2 + 1 \cdot \bar{x}_1 \cdot \bar{x}_2 + 1 \cdot \bar{x}_1 \cdot \bar{x}_2 + 1 \cdot \bar{x}_1 \cdot \bar{x}_2.
\]

The defining table for the function \( f(x_1, x_2) \) is shown below:

<table>
<thead>
<tr>
<th>( x_2 )</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 )</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

In order to obtain the coefficients of the fixed polarity canonical form we construct the matrix \( F^2 \), and then multiply it by \( T^2 \).

\[
C = F^2 \odot T^2 = \begin{bmatrix}
112 & 010 & 021 \\
121 & 100 & 210 \\
211 & 001 & 102 \\
010 & 021 & 112 \\
100 & 210 & 121 \\
001 & 102 & 211 \\
021 & 112 & 010 \\
210 & 121 & 100 \\
102 & 211 & 001
\end{bmatrix} \odot \begin{bmatrix}
122 & 211 & 211 \\
010 & 020 & 020 \\
001 & 002 & 002 \\
000 & 122 & 000 \\
000 & 010 & 000 \\
000 & 001 & 000 \\
000 & 000 & 122 \\
000 & 000 & 010 \\
000 & 000 & 001
\end{bmatrix} = \begin{bmatrix}
101 & 212 & 220 \\
110 & 012 & 111 \\
222 & 112 & 202 \\
010 & 011 & 121 \\
122 & 102 & 021 \\
001 & 120 & 221 \\
021 & 110 & 022 \\
221 & 222 & 201 \\
121 & 101 & 210
\end{bmatrix}
\]

The fourth, sixth and seventh rows of \( C \) have the largest number of zero-valued coefficients (three). Hence, polarities \( k = 3, 5 \) and \( 6 \) are the polarities for which the canonical form of the function \( f(x_1, x_2) \) has a minimal number of non-zero valued coefficients. For example, for \( k = 6 \), i.e. \((k_2k_1) = (20)\), the function \( f(x_1, x_2) \) has the following canonical form:

\[
f(x_1, x_2) = c_1 \cdot x_1^{\frac{1}{2}k_1} \cdot x_2^{\frac{1}{2}k_1} \oplus c_2 \cdot x_1^{\frac{1}{2}k_1} \cdot x_2^{\frac{1}{2}k_1} \oplus c_3 \cdot x_1^{\frac{1}{2}k_1} \cdot x_2^{\frac{1}{2}k_1} \oplus c_4 \cdot x_1^{\frac{1}{2}k_1} \cdot x_2^{\frac{1}{2}k_1} \oplus c_5 \cdot x_1^{\frac{1}{2}k_1} \cdot x_2^{\frac{1}{2}k_1} \oplus c_6 \cdot x_1^{\frac{1}{2}k_1} \cdot x_2^{\frac{1}{2}k_1} \oplus c_7 \cdot x_1^{\frac{1}{2}k_1} \cdot x_2^{\frac{1}{2}k_1} \oplus c_8 \cdot x_1^{\frac{1}{2}k_1} \cdot x_2^{\frac{1}{2}k_1} \\
= 0 \oplus 2 \cdot x_1^{\frac{1}{2}k_1} \cdot x_2^{\frac{1}{2}k_1} \oplus 2 \cdot x_1^{\frac{1}{2}k_1} \cdot x_2^{\frac{1}{2}k_1} \oplus 0 \cdot x_1^{\frac{1}{2}k_1} \cdot x_2^{\frac{1}{2}k_1} \oplus 0 \cdot x_1^{\frac{1}{2}k_1} \cdot x_2^{\frac{1}{2}k_1} \\
= x_1 \oplus \bar{x}_1 \cdot x_2 \oplus \bar{x}_1 \cdot \bar{x}_2 \oplus \bar{x}_1 \cdot \bar{x}_2 \\
= x_1 \oplus \bar{x}_1 \cdot x_2 \oplus \bar{x}_1 \cdot \bar{x}_2 \oplus \bar{x}_1 \cdot \bar{x}_2 \\
= \bar{x}_1 \oplus x_2 \oplus \bar{x}_1 \cdot x_2 \oplus \bar{x}_1 \cdot \bar{x}_2 \oplus \bar{x}_1 \cdot \bar{x}_2 .
\]
CHAPTER 3. A CANONICAL FORM OF MVL FUNCTIONS

3.8 Conclusion

This chapter introduces a canonical form for multiple-valued functions, based on the operations of addition modulo \( m \), minimum and the set of all literal operators. In the two-valued case this form is equivalent to the fixed polarity Reed-Muller canonical form.

One advantage of the new generalization over other generalizations of the Reed-Muller canonical form previously proposed is that it is defined for \( m \) being any positive integer, whereas other representations are applicable only for the case of \( m \) being a prime or a power of a prime number.

Another advantage is that, from an implementation point of view, the MIN operation is normally much easier to implement than the multiplication modulo \( m \) operation. In Appendix A we show a CMOS transistor-level realization of the basic gates of the algebra \( C \) and a simulation of these gates using the HSPICE program. As follows from the Appendix, MIN can be implemented as a current-mode CMOS MVL circuit using only 5 transistors. Moreover, this implementation is independent of \( m \), i.e. the number of transistors does not increase with increasing values of \( m \). On the other hand, the implementation of multiplication modulo \( m \) is much more complex and always depends on \( m \). For example, for \( m = 3 \), a current-mode CMOS MVL circuit, realizing a multiplication modulo \( m \), consists of 16 transistors [64].

Obviously, this implementation advantage would lead to decreasing the overall complexity of the realization of an \( m \)-valued function only if the number of terms and number of literals per term in the new canonical form is smaller than the corresponding numbers in other representations for the same function. So, further research needs to be done to estimate the complexity of the new canonical form by evaluating the number of terms and number of literals per term in the expression for a given function as compared to the corresponding numbers in other representations.
Chapter 4

AND-OR-XOR Minimization of Boolean Functions

This chapter considers the problem of logic minimization of Boolean functions, where the target is to find a minimal expansion for the function in terms of \( \text{AND}, \text{OR}, \text{XOR} \) and \( \text{NOT} \) operations, with certain restrictions on the structure of the expansion. This is the only chapter of the dissertation where the problem is restricted to the two-valued case only. The problem considered is very hard and so far no efficient solution has been found even for this simple case of \( m = 2 \).

In section 4.1 we give some background on logic minimization and motivate our interest in AND-OR-XOR minimization. Section 4.2 describes notations and definitions used in this chapter. In section 4.3 we prove the upper bound on the number of AND-terms in the expansion we consider. Section 4.4 presents an algorithm for AND-OR-XOR minimization of Boolean functions with a worst-case time complexity \( O(N^3) \), where \( N = 2^n \) and \( n \) is the number of variables in the function. In section 4.5 we describe the experiments we performed to evaluate the algorithm. The chapter concludes with suggestions of work following from this research.

Some of the results in this chapter are contained in [20] and [24].
4.1 Logic minimization

Logic minimization plays an important role in logic synthesis. The resulting layout area and delay of the synthesized circuit usually depends on this step to a significant degree. The purpose of logic minimization is to obtain a specified algebraic expansion for the function, which is minimal with respect to some criteria for minimality. The criteria might be to make the number of terms in the expansion and the number of variables in these terms as small as possible. The simplified expansion can be considered as a structural description of the logic circuit, providing the list of required basic gates together with a description of their interconnection. If the technology allows the building of gates realizing the operations of the expansion, then the logic circuit can be synthesized directly from the algebraic expansion.

Logic minimization techniques can be classified with respect to the set of operations used in the algebraic expansion and with respect to the number of levels in the logic circuit that is targeted. Number of levels in the logic circuit equals the maximum number of gates cascaded in series between a circuit input and the output. With respect to the number of levels, the minimization techniques are classified into two-level and multiple-level minimizers. With respect to the set of operations, the techniques are distinguished as AND-OR, AND-XOR or AND-OR-XOR. The first category seeks a minimal expansion for the function in terms of AND, OR and NOT operations. the second looks for a minimal expansion in terms of AND, XOR and NOT operations. the third, most general one, attempts to find a minimal expansion composed of AND, OR, XOR and NOT.

AND-OR logic minimization has been extensively studied for many years. Espresso [5] is an example of a two-level AND-OR minimizer. MISII [4] is a popular package for multi-level AND-OR minimization.

AND-XOR and AND-OR-XOR logic minimizations are a less-developed area. Partly, this is due to the fact that the gate realizing the XOR operation has a prohibitive cost in some technologies. One of the reasons for this is that in these
technologies an XOR gate takes more chip area than an OR or an AND gate. Another reason is a larger delay for XOR gates. However, in newer technologies like Field Programmable Gate Arrays (FPGA) the delay and area of all gates are equal. For example, in the Xilinx look-up table type FPGA's, the basic combinational block can realize any function of up to five variables with the same area and delay [66]. Similarly, among the sea-of-gate style FPGA's, the Cli606 from Concurrent Logic Inc. include a 2-input XOR gate as its basic granularity block [61]. Also, in PlusLogic three-level FPGA, the first two levels are implemented by a Programmable Logic Array (PLA) and the third by a set of logic expanders [37]. Each logic expander can be programmed to realize any function of two variables, so programming it to implement the XOR brings no disadvantage as compared to AND or OR.

The introduction of the new FPGA technology brought the attention of many researchers to the development of efficient algorithms for logic minimization including XOR in the set of basic operations. A number of algorithms addressing AND-XOR minimization were reported, including those in [28], [38], [53], [54] and [61]. Fewer results are known on AND-OR-XOR minimization. One such result is due to Chattopadhyay et al. [8]. Their algorithm integrates AND-XOR and AND-OR minimization techniques. It performs a subsequent decomposition of the function (Shannon decomposition (2.1), negative or positive decomposition (3.6)), and, at every stage, evaluates which type of decomposition should be applied to the resulting subfunctions. The result of the algorithm is a multi-level logic circuit. The algorithm has been shown to outperform the popular package for multi-level AND-OR minimization MISII [4].

Another algorithm for AND-OR-XOR minimization is developed in [51]. It first generates a minimal expansion of the function in terms of XOR and AND operations (XOR of AND-terms). Some of the XOR's are then converted into OR's, and subsequently, a graph coloring technique is used for the minimization of AND-terms in the final expansion. This technique generates a three-level circuit implementing
XOR of two AND-OR expansions (which are the OR of AND-terms).

In this chapter, we present a new heuristic algorithm for AND-OR-XOR minimization. As with any heuristic algorithm, ours does not guarantee that a minimal solution is found, but usually obtains a nearly-minimal one. However, as the experimental results show, the algorithm does have satisfactory performance for some common benchmark functions. We also prove an upper bound on the number of AND-terms in the expansion generated by the algorithm. Such a bound is useful for estimating the size of the FPGA, as well as for asserting the minimality of the solution obtained by the heuristic algorithm.

4.2 Notation and definitions

Throughout this chapter, \( f(x_1, \ldots, x_n) \) denotes a Boolean function \( f : B^n \to Y \), where \( B = \{0, 1\} \) and \( Y = \{0, 1, *\} \), with * denoting a don't care value. A cube is an n-tuple of 0's, 1's or *'s. If a cube contains no *'s, then it is termed a vertex. A minterm represents a point in the domain \( B^n \) of the function. The on-set \( T \), the don't care-set \( D \) and the off-set \( F \) of \( f \) are the sets of cubes that are mapped by \( f \) to 1, *, and 0, respectively. If \( D = \emptyset \), the function is called completely specified, otherwise it is incompletely specified. A realization of an incompletely specified function \( g : B^n \to Y \) is any completely specified function \( f : B^n \to B \) such that for every n-tuple \( a \in B^n \), if \( g(a) \in B \), then \( f(a) = g(a) \).

The representation of a function in terms of cubes can be mapped into an algebraic representation of the function. In Boolean algebra, an algebraic representation of \( f \) is a Boolean expression that evaluates to 1 for all elements of \( T \), evaluates to 0 for all elements of \( F \), and evaluates to either 1 or 0 for all elements of \( D \). We use the algebraic representation of functions in deriving the upper bound in section section 4.3 as well as for discussion on the algorithm in section section 4.4. The cubical representation of functions is used to carry out the computation in the algorithm.

In an algebraic representation of a function, a literal is a variable or its comple-
CHAPTER 4. AND-OR-XOR MINIMIZATION OF BOOLEAN FUNCTIONS

A *product-term* is an AND of zero or more literals, such that no two of them are complements of each other. A product-term is the algebraic equivalent of a cube. A *minterm* of an $n$-variable function is a product-term consisting of $n$ literals. A minterm is the algebraic equivalent of a vertex.

A **AND-OR** expansion is an OR of product-terms. AND-OR expansion is often referred to as *sum-of-products* expansion, but we avoid this name to refrain from confusion with modulo $m$ sum-of-products form, cited in several other chapters of the dissertation. An **AND-XOR** expansion is an XOR of product-terms.

An **AND-OR-XOR** expansion is the XOR of two AND-OR expansions. A minimal AND-OR-XOR expansion is the expansion with the smallest number of product-terms. For example, consider the function $f(x_1, x_2, x_3, x_4)$ shown in Figure 4.1. Its minimal AND-OR-XOR expansion consists of 4 product-terms, namely:

$$f(x_1, x_2, x_3, x_4) = (x_1'x_2 + x_3x_4) \oplus (x_1'x_2' + x_3'x_4')$$

Figure 4.1: Karnaugh map of the function from the example.

The minimal AND-OR expansion for the same function has 8 product terms:

$$f(x_1, x_2, x_3, x_4) = x_1'x_2x_3' + x_1'x_2x_4 + x_1x_2x_3 + x_1x_2x_4 + x_1x_2'x_3 + x_1x_2'x_4 + x_1x_3x_4 + x_1x_3x_4'$$

and the minimal AND-XOR expansion (in which both complemented and uncomplemented forms of a variable are used) has 6 product terms:

$$f(x_1, x_2, x_3, x_4) = x_1x_3 \oplus x_2x_4 \oplus x_1'x_3x_4' \oplus x_1'x_2x_3' \oplus x_1x_2'x_4' \oplus x_1x_2x_4'.$$
CHAPTER 4. AND-OR-XOR MINIMIZATION OF BOOLEAN FUNCTIONS

4.3 Upper bound on the number of product-terms in the AND-OR-XOR expansion

In this section we present a theorem which gives an upper bound on the number of product-terms in the AND-OR-XOR expansion. The proof of the theorem is based on the following property.

**Property 4.1** Let $f_1, f_2, g_1, g_2$ be Boolean functions. The following equations hold:

(a) $x'(f_1 \oplus f_2) + x(g_1 \oplus g_2) = (x'f_1 + xg_1) \oplus (x'f_2 + xg_2)$

(b) $x'(f_1 \oplus f_2) + x(g_1 \oplus g_2) = (x'f_1 + xg_2) \oplus (x'f_2 + xg_1)$

**Proof (a):** Let $x = 0$. Then the left hand side is $(f_1 \oplus f_2) + 0 = f_1 \oplus f_2$. And the right hand side is $(f_1 + 0) \oplus (f_2 + 0) = f_1 \oplus f_2$. Similarly, for $x = 1$, both left and right sides equal $g_1 \oplus g_2$. For the case (b) the proof is similar.

Let $p_i$ and $q_i, i > 0$, denote arbitrary product-terms involving some of the variables $x_1, \ldots, x_n$ or their complements. For example, the function $f(x_1, x_2, x_3) = x_1x_2 + x'_1x_2x'_3$ can be written as $f(x_1, x_2, x_3) = p_1 + p_2$, with $p_1 = x_1x_2$ and $p_2 = x'_1x_2x'_3$.

**Theorem 4.2** Every Boolean function $f(x_1, \ldots, x_n)$ of $n$ variables ($n \geq 4$) can be expanded using at most $5 \cdot 2^{n-4}$ product-terms as:

$$f(x_1, \ldots, x_n) = (p_1 + p_2 + \ldots + p_i) \oplus (p_{i+1} + p_{i+2} + \ldots + p_j),$$

for some $i$ in $1 \leq i \leq j$.

**Proof:** By induction on $n$.

1) Let $n = 4$. By exhaustive search through the 402 PN-equivalence classes \(^1\) of

---

\(^1\)In the PN classification all functions which differ only by some permutation of the input variables and/or by complementation of one or more of the input variables are considered as being in the same classification entry [30]. An AND-OR-XOR expansion is an invariant over a PN-class.
CHAPTER 4. AND-OR-XOR MINIMIZATION OF BOOLEAN FUNCTIONS

Boolean functions of 4-variables we can establish that each function can be expressed using at most 5 product-terms. Hence, for $n = 4$ the expansion exists and $j \leq 5$.

2) Hypothesis: Assume the result holds for functions of $n$ or less variables. Using Shannon decomposition (2.1), any Boolean function of $n+1$ variables can be expanded as:

$$f(x_1, \ldots, x_{n+1}) = x'_{n+1}f(x^0_{n+1}) + x_{n+1}f(x^1_{n+1}).$$

Recall that $f(x^0_{n+1})$ and $f(x^1_{n+1})$ denote subfunctions of the function $f(x_1, \ldots, x_{n+1})$ for which $x_{n+1} = 0$ and $x_{n+1} = 1$, respectively.

According to the inductive hypothesis, these subfunctions can be expanded as:

$$f(x^0_{n+1}) = (p_1 + p_2 + \ldots + p_l) \oplus (p_{i+1} + p_{i+2} + \ldots + p_j)$$

$$f(x^1_{n+1}) = (q_1 + q_2 + \ldots + q_l) \oplus (q_{k+1} + q_{k+2} + \ldots + q_t).$$

with $j \leq 5 \cdot 2^{n-4}$, $l \leq 5 \cdot 2^{n-4}$, and for some $i$ and $k$ such that $1 \leq i \leq j$, $1 \leq k \leq l$. Then:

$$f(x_1, \ldots, x_{n+1}) = x'_{n+1}f(x^0_{n+1}) + x_{n+1}f(x^1_{n+1})$$

$$= x'_{n+1} ((p_1 + \ldots + p_l) \oplus (p_{i+1} + \ldots + p_j)) +$$

$$+ x_{n+1} ((q_1 + \ldots + q_l) \oplus (q_{k+1} + \ldots + q_t)) \quad \text{(substitution)}$$

$$= (x'_{n+1}(p_1 + \ldots + p_l) + x_{n+1}(q_1 + \ldots + q_l)) \oplus$$

$$\oplus (x'_{n+1}(p_{i+1} + \ldots + p_j) + x_{n+1}(q_{k+1} + \ldots + q_t)) \quad \text{(Property 4.1)}$$

$$= (x'_{n+1}p_1 + \ldots + x'_{n+1}p_l + x_{n+1}q_1 + \ldots + x_{n+1}q_l) \oplus$$

$$\oplus (x'_{n+1}p_{i+1} + \ldots + x'_{n+1}p_j + x_{n+1}q_{k+1} + \ldots + x_{n+1}q_t) \quad \text{(Distributivity of \cdot over +)}$$

Since $i \leq 5 \cdot 2^{n-4}$ and $k \leq 5 \cdot 2^{n-4}$, therefore $i+k+(j-i)+(l-k) = j+l \leq 5 \cdot 2^{(n+1)-4}$.

$\square$
By a search through the PN-equivalence classes of Boolean functions of 5-variables, Debnath and Sasao in [15] have shown that none of the 5-variable functions requires more than 9 product-terms in its minimal AND-OR-XOR expansion. This result allows them to change the basic step of the inductive proof of the Theorem 4.2, leading to a tighter bound of $9 \cdot 2^{n-5}$, for $n \geq 5$. The same authors have also recently shown in [16] that if two AND-OR expansions, $A_1$ and $A_2$, are allowed to have common product-terms, then the upper bound on the number of products in AND-OR-XOR expansion of type $A_1 \oplus A_2$ is $15 \cdot 2^{n-6}$, for $n \geq 6$.

Theorem 4.2 shows that an AND-OR-XOR has a smaller upper bound on the number of product-terms than the bound $2^{n-1}$ of the AND-OR expansion and $3 \cdot 2^{n-3}$ for the AND-XOR expansion [52]. This means that, for some functions, the AND-OR-XOR expansion consists of fewer product-terms that AND-OR and AND-XOR expansions, implying the existence of a more economical circuit implementation for the function. However, this advantage can be utilized only if an efficient algorithm for computing a minimal AND-OR-XOR expansion exists. In the next section we present one such algorithm.

4.4 An algorithm for minimizing AND-OR-XOR expansions

The problem of finding a minimal AND-OR-XOR expansion can be formulated as follows:

\textit{Given a function } $f = (T, D, F)$, \textit{find two AND-OR expansions } $g_1$ \textit{and } $g_2$ \textit{such that } $g_1 \oplus g_2$ \textit{realizes } $f$ \textit{whenever } $f$ \textit{is defined and the total number of product-terms in } $g_1$ \textit{and } $g_2$ \textit{is minimized.}

The basic steps of our algorithm for solving this problem are as follows:
Algorithm for minimizing AND-OR-XOR expansions (AOXMIN)

input: \( f = (T, D, F) \) of \( n \) variables, an integer \( N_{\text{iter}} \) (linear in \( n \))

output: AND-OR expansions \( g_1 \) and \( g_2 \) such that \( g_1 \oplus g_2 \) realizes \( f \) and the total number of product-terms of \( g_1 \) and \( g_2 \) is minimized.

1. Use Espresso to minimize \( f \).
2. Fill \( ON \) list and \( OFF \) list with the \( T \) and \( F \) cubes from the resulting function.
3. Initialize \( g_{1,\text{best}} \) and \( g_{2,\text{best}} \).
4. Using \( \text{DivideEqClasses()} \), divide the cubes from \( ON \) list into equivalence classes.
5. Using \( \text{SelectPartitioning()} \), group the resulting equivalence classes into two sets, \( T_1 \) and \( T_2 \).
6. Construct \( g_{1,\text{init}} = (T_1, F, T_2) \) and \( g_{2,\text{init}} = (T_2, F, T_1) \).
7. Starting from \( g_{1,\text{init}} \), invoke \( \text{SpecifyBoth()} \) to determine which don't cares in \( g_{1,\text{init}} \) and \( g_{2,\text{init}} \) should be specified to 1 so that the total number of product-terms in both functions is minimized.
8. Choose which of the pairs of functions \( (g_1, g_2), (g_1, g'_2), (g'_1, g_2) \) or \( (g'_1, g'_2) \) has the smallest number of product-terms, and save it.
9. Repeat steps 7 and 8 starting from \( g_{2,\text{init}} \).
10. Repeat steps 5 - 9 for \( N_{\text{iter}} \) partitionings.
11. Repeat steps 4 - 10 starting from the \( OFF \) list.

The implementation of the algorithm uses well-known programming methods. Functions are represented dynamically by lists of cubes. Each cube is represented by a structure declared as follows:
typedef struct ListofCubes {
    long int min: /* minimum minterm value */
    long int max: /* maximum minterm value */
    int class: /* equival. class to which the cube belongs */
    struct ListofCubes *next: /* pointer to the next cube */
} ListofCubes;

This approach gives very fast performance of the basic operations required in the algorithm. Our current implementation only accommodates functions up to 32 variables. However, it can be modified to handle larger functions by storing each of the min and max values in two (or more) full words (32 bits), instead of just using one.

In the subsequent sections we explain the basic steps in more detail.

4.4.1 Dividing the cubes into equivalence classes

Our heuristics are based on the observation that product-terms in a minimal AND-OR expansion of a function may give some information about how to partition its on-set $T$ into two sets. To explain this intuition, consider the Boolean function from the previous example (Figure 4.1). The minimal AND-OR expansion of this function consists of 8 product-terms, namely:

$$f(x_1, \ldots, x_4) = x_1'x_2x_3' + x_1'x_2x_4 + x_1'x_3x_4 + x_2x_3'x_4 + x_1x_2'x_4' + x_1x_2x_3 + x_1x_3x_4' + x_2'x_3x_4'$$

These product-terms are shown in Figure 4.2.

Suppose that we want to represent this function as $f = g_1 \oplus g_2$ and we put an additional constraint that each of the 8 product-terms from the AND-OR expansion is entirely contained in either $g_1$ or $g_2$. Then we can easily see that if two product-terms, $p_i$ and $p_j$ have minterms in common, i.e. if $p_i \cdot p_j \neq 0$, then they both have to belong to either $g_1$ or $g_2$, since otherwise $g_1 \oplus g_2 = 0$ for the common minterms. For example, $p_1 = x_1'x_2x_3'$ and $p_2 = x_1'x_3'x_4$ have minterms in common, since $x_1'x_2x_3'$.
CHAPTER 4. AND-OR-XOR MINIMIZATION OF BOOLEAN FUNCTIONS

If $p_1$ is in $g_1$ and $p_2$ is in $g_2$, when $x'_1 x'_2 x'_3 x'_4$ is not in $g_1 \supseteq g_2$, and therefore $g_1 \supseteq g_2 \neq f$. This constraint allows us to reduce the search-space for possible partitionings. As shown below, this leads to a reduction in the time-complexity of the algorithm from $O(2^N)$ to $O(N^3)$.

Since the algorithm performs partitionings of product-terms, not single minterms, its first step is obtaining a minimal AND-OR expansion $\hat{f}$ for $f$. The minimization is carried out with Espresso [5]. We couldn’t find out the actual time complexity of a single run of Espresso (we even contacted the authors), but it appears to be $O(N)$, and we assume here this estimate. After minimization, two lists, $ON\text{\_list}$ and $OFF\text{\_list}$, are formed from the $T$ and $F$ cubes of $f$.

The next step is to initialize $g1\_best$ and $g2\_best$. If the number of cubes in $ON\text{\_list}$ is less than the number of cubes in $OFF\text{\_list}$, then $g1\_best$ is initialized to $\hat{f}$ and $g2\_best$ is initialized to the constant zero function. otherwise $g1\_best = \hat{f}'$ and $g2\_best = 1$. In this way, for a single function, the number of product-terms generated by AOXMIN is never larger than the one that is produced by Espresso.

In the next step, the procedure $\text{DivideEqClasses()}$ is called to divide the $ON\text{\_list}$ cubes into equivalence classes. A sketch of the code is shown in Figure 4.3.

The cubes $c_i$ are divided by $\text{DivideEqClasses()}$ into equivalence classes w.r.t. the equivalence relation $R := R_1^+$. where $R_1^+$ is the transitive closure of $R_1$, and $R_1$ is defined by

Figure 4.2: Map of the example function.

$x'_1 x'_3 x'_4 = x'_1 x'_2 x'_3 x'_4$. If $p_1$ is in $g_1$ and $p_2$ is in $g_2$, when $x'_1 x'_2 x'_3 x'_4$ is not in $g_1 \supseteq g_2$, and therefore $g_1 \supseteq g_2 \neq f$. This constraint allows us to reduce the search-space for possible partitionings. As shown below, this leads to a reduction in the time-complexity of the algorithm from $O(2^N)$ to $O(N^3)$.

Since the algorithm performs partitionings of product-terms, not single minterms, its first step is obtaining a minimal AND-OR expansion $\hat{f}$ for $f$. The minimization is carried out with Espresso [5]. We couldn’t find out the actual time complexity of a single run of Espresso (we even contacted the authors), but it appears to be $O(N)$, and we assume here this estimate. After minimization, two lists, $ON\text{\_list}$ and $OFF\text{\_list}$, are formed from the $T$ and $F$ cubes of $\hat{f}$.

The next step is to initialize $g1\_best$ and $g2\_best$. If the number of cubes in $ON\text{\_list}$ is less than the number of cubes in $OFF\text{\_list}$, then $g1\_best$ is initialized to $\hat{f}$ and $g2\_best$ is initialized to the constant zero function. otherwise $g1\_best = \hat{f}'$ and $g2\_best = 1$. In this way, for a single function, the number of product-terms generated by AOXMIN is never larger than the one that is produced by Espresso.

In the next step, the procedure $\text{DivideEqClasses()}$ is called to divide the $ON\text{\_list}$ cubes into equivalence classes. A sketch of the code is shown in Figure 4.3.

The cubes $c_i$ are divided by $\text{DivideEqClasses()}$ into equivalence classes w.r.t. the equivalence relation $R := R_1^+$. where $R_1^+$ is the transitive closure of $R_1$, and $R_1$ is defined by
DivideEqClasses(list.name) {
  input: a list of cubes
  output: for each cube the field "class" is filled with the number of the equivalence
          class to which it belongs

  for (each cube a from the first to the last in the list list.name) {
    for (each cube b from the next after a to the last in the list list.name) {
      if (a and b intersect) {
        if (b.class is not labeled)
          b.class = a.class
        else {
          b.class = a.class
          find all cubes with the same class as b and set them to a.class
        }
      }
    }
  }
}

Figure 4.3: Implementation of the subroutine DivideEqClasses().

\[(c_i, c_j) \in R_1 \text{ iff } (c_i \cap c_2 \neq \emptyset)\]

where \(\emptyset\) denotes an empty set and \(\cap\) denotes the intersection of cubes. defined by the
following table:

<table>
<thead>
<tr>
<th>(\cap)</th>
<th>(c_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 1 *</td>
</tr>
<tr>
<td>(c_2)</td>
<td>1 (\emptyset) 1 1</td>
</tr>
<tr>
<td>*</td>
<td>0 1 (\emptyset)</td>
</tr>
</tbody>
</table>

Two cubes are in relation \(R\) either when they intersects, or then they are connected
through a chain of intersecting cubes. The equivalence classes of \(R\) form partition of
the set of all cubes into connected chains of cubes. Since our algorithm requires a
cube to be entirely included in either \(g_1\) or \(g_2\), it follows that each equivalence class
of cubes must be entirely contained in either \(g_1\) or \(g_2\).
The procedure DivideEqClasses() takes a list of cubes as its input, computes which cubes are connected, and, for each cube, fills the field "class" with the number of the equivalence class to which it belongs. In the main program, it is run twice - first starting from ON list and then starting from OFF list. The procedure has time-complexity $O(k^3)$ for a list of $k$ elements, which are the product-terms $p_i$ in our case. Since the upper bound on the number of product-terms in the minimal AND-OR expansion is $2^{n-1}$, the worst-case time complexity of DivideEqClasses() is $O((2^{n-1})^3) = O(N^3)$. Although in terms of "big-oh" complexity this subroutine is the most time-consuming one, in the experimental results section it is shown that for practical functions it is quite fast.

4.4.2 Obtaining $T_1$ and $T_2$

If DivideEqClasses() results in more than two equivalence classes, then they should be grouped into two sets to get $T_1$ and $T_2$, which are the initial on-sets for $g_1$ and $g_2$. This is done by a procedure SelectPartitioning().

The number of possible ways to group $k$ classes into two sets is $N_{all} = 2^{k-1} - 1$. If $k$ is large, trying all possible partitionings may result in an unreasonably long computational time. To avoid this, our program takes as a parameter an integer $N_{iter}$, indicating the number of partitionings we are willing to try, which we restrict to be linear in the number of variables $n$. If $N_{iter}$ is larger than or equal to $N_{all}$, then SelectPartitioning() successively tries all possible partitionings. Otherwise, it generates $N_{iter}$ number of randomly chosen partitionings. As the experimental results section shows, for most practical functions 20 or less iterations are sufficient to obtain good results.

The procedure SelectPartitioning() is called only if the number of equivalence classes, obtained by DivideEqClasses(), is more than one. Otherwise, if $J$ has only one equivalence class, as for example the function shown in Figure 4.4, then AOXMN returns the functions $g_1\text{.best}$ and $g_2\text{.best}$ as initialized in step 3 of the algorithm. It
is our experience that for the functions with only one equivalence class introducing an XOR does not bring any advantage in terms of the number of product-terms.

### 4.4.3 Constructing $g_1\_init$ and $g_2\_init$

After partitioning the set $T$ into two subsets $T_1$ and $T_2$, the functions $g_1\_init = (T_1, D_1, F_1)$ and $g_2\_init = (T_2, D_2, F_2)$ are constructed, so that:

- $T_1$ and $T_2$ are the on-sets obtained after partitioning of the on-set $T$ of $f$.
- $D_1 = F$ and $D_2 = F$ are the don't care sets, i.e. the off-set of function $f$ is the don't care-set for $g_1$ and $g_2$.
- $F_1 = T_2$ and $F_2 = T_1$ are the off-sets, i.e. the on-set of $g_1$ is the off-set of $g_1$, and vice versa.

Figure 4.5 shows $g_1\_init$ and $g_2\_init$ for the function from Figure 4.1. Here the number of equivalence classes equals two, and therefore there is only one way to choose $T_1$ and $T_2$.

### 4.4.4 Determining common don't cares in $g_1\_init$ and $g_2\_init$

After $g_1\_init$ and $g_2\_init$ are obtained, the next step is to determine which don't cares should be specified to be 1 so that the total number of product-terms in AND-
CHAPTER 4. AND-OR-XOR MINIMIZATION OF BOOLEAN FUNCTIONS

Figure 4.5: Functions $g_{\text{init}}$ and $g'_{\text{init}}$ for the function from the example.

OR expansion of both functions is minimized. For this, we invoke the subroutine SpecifyBoth(), shown in Figure 4.6.

SpecifyBoth($T_1, D_1, F_1, T_2, D_2, F_2$) {
input: two incompletely specified functions $g_{\text{init}} = (T_1, D_1, F_1)$ and $g'_{\text{init}} = (T_2, D_2, F_2)$.
output: two AND-OR expansions $g_1$ and $g_2$, realizing $g_{\text{init}}$ and $g'_{\text{init}}$ correspondently.

$$g_1 = \text{Espresso} \ (T_1, D_1, F_1);$$
$$T'_1 = g_1 - T_1;$$
$$T_2 = T_2 \cup T'_1;$$
$$F_2 = F_2 \cup (D_2 - T'_1);$$
$$g_2 = \text{Espresso} \ (T_2, D_2, F_2);$$
return ($g_1, g_2$): }

Figure 4.6: Implementation of the subroutine SpecifyBoth().

First, the function $g_{\text{init}} = (T_1, D_1, F_1)$ is minimized using Espresso, giving $g_1$. Then, it is determined which don't cares from $g_{\text{init}}$ were specified to 1 in $g_1$ by computing the difference $T'_1 = g_1 - T_1$. These don't cares are specified to 1 in $g'_{\text{init}}$ and all other don't cares in $g'_{\text{init}}$ are set to 0. The resulting function is minimized using Espresso.

In the main program, the procedure SpecifyBoth() is run twice - first, starting from $g_{\text{init}}$ and next starting from $g'_{\text{init}}$. Each time the result with the smaller number of product-terms in the pairs of functions ($g_1, g_2$), ($g'_1, g'_2$), ($g'_1, g_2$) or ($g_1, g'_2$)
is compared to the "best" result obtained so far and is saved in case it is less than
the "best". All the subroutines in the steps from 5 to 9 are $O(N)$, and therefore,
if the parameter $N_{\text{iter}}$ is chosen to be linear in $n$, the time-complexity of the loop
is $O(N \log N)$. The overall complexity of the main procedure is determined by the
highest degree of $O$ among the subroutines used, i.e. by $\text{DivideEqClasses}()$, and
thus is $O(N^2)$.

4.4.5 Multiple-output problems

Most digital circuits have multiple outputs. In our current implementation, we con­
sider a $k$-output circuit as consisting of $k$ separate single-output circuits. We first run
AOXMIN for each of the $k$ functions describing single-output circuits to obtain their
$g_{1,\text{best}}$ and $g_{2,\text{best}}$. Then we apply Espresso to the resulting $2k$ functions to iden­
tify common product-terms. This exploits common product-terms at least to some
degree. Clearly, treating $k$ functions simultaneously throughout the algorithm would
lead to better results.

4.5 Experimental Results

We have implemented the algorithm described in the previous section and have ap­
plied it to a set of benchmark functions. The benchmark functions are taken from
http://www.cbl.ncsu.edu/pub/benchmark_dirs/LGSynth91/twolexamples/. We have
compared the results of our program with the performance of the two-level AND­
OR minimizer Espresso [5] (with and without output phase optimization) as well as
with the results reported in [51]. AOXMIN and Espresso were run on a Sun SPARC
20 operating at 50 MHz with 64 MB RAM main storage.

Table 4.1 shows the number of product-terms in the resulting functions and the
time taken in seconds in columns $pr.$ and $t.$ respectively. The time is user time
measured using the UNIX system command $time$. Column 8 shows the number of
product-terms obtained by the AND-OR-XOR minimizer from [51] for the bench¬
### Table 4.1: Benchmark results.

<table>
<thead>
<tr>
<th>Example function</th>
<th>n</th>
<th>m</th>
<th>Espresso [5]</th>
<th>Alg. [51]</th>
<th>AOXMIN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pr.</td>
<td>t.sec</td>
<td>pr.</td>
<td>t.sec</td>
<td>pr.</td>
</tr>
<tr>
<td>5xp1</td>
<td>7</td>
<td>10</td>
<td>65</td>
<td>0.28</td>
<td>64</td>
</tr>
<tr>
<td>9sym</td>
<td>9</td>
<td>1</td>
<td>86</td>
<td>0.51</td>
<td>86</td>
</tr>
<tr>
<td>alu4</td>
<td>14</td>
<td>8</td>
<td>575</td>
<td>29.39</td>
<td>359</td>
</tr>
<tr>
<td>b12</td>
<td>15</td>
<td>9</td>
<td>43</td>
<td>0.69</td>
<td>29</td>
</tr>
<tr>
<td>bw</td>
<td>5</td>
<td>28</td>
<td>22</td>
<td>0.62</td>
<td>22</td>
</tr>
<tr>
<td>clip</td>
<td>9</td>
<td>5</td>
<td>120</td>
<td>1.50</td>
<td>120</td>
</tr>
<tr>
<td>con1</td>
<td>7</td>
<td>2</td>
<td>9</td>
<td>0.1</td>
<td>8</td>
</tr>
<tr>
<td>cordic</td>
<td>23</td>
<td>2</td>
<td>914</td>
<td>123.34</td>
<td>155</td>
</tr>
<tr>
<td>duke2</td>
<td>22</td>
<td>29</td>
<td>86</td>
<td>0.93</td>
<td>36</td>
</tr>
<tr>
<td>ex1010</td>
<td>10</td>
<td>10</td>
<td>284</td>
<td>42.24</td>
<td>279</td>
</tr>
<tr>
<td>inc</td>
<td>7</td>
<td>9</td>
<td>30</td>
<td>0.18</td>
<td>28</td>
</tr>
<tr>
<td>misex1</td>
<td>8</td>
<td>7</td>
<td>12</td>
<td>0.04</td>
<td>12</td>
</tr>
<tr>
<td>misex2</td>
<td>25</td>
<td>18</td>
<td>28</td>
<td>0.07</td>
<td>28</td>
</tr>
<tr>
<td>misex3c</td>
<td>14</td>
<td>14</td>
<td>690</td>
<td>41.86</td>
<td>189</td>
</tr>
<tr>
<td>misex3</td>
<td>14</td>
<td>14</td>
<td>197</td>
<td>13.00</td>
<td>199</td>
</tr>
<tr>
<td>rd53</td>
<td>5</td>
<td>3</td>
<td>31</td>
<td>0.05</td>
<td>22</td>
</tr>
<tr>
<td>rd73</td>
<td>7</td>
<td>3</td>
<td>127</td>
<td>0.44</td>
<td>93</td>
</tr>
<tr>
<td>rd84</td>
<td>8</td>
<td>4</td>
<td>255</td>
<td>1.65</td>
<td>186</td>
</tr>
<tr>
<td>sao2</td>
<td>10</td>
<td>4</td>
<td>58</td>
<td>0.22</td>
<td>37</td>
</tr>
<tr>
<td>squar5</td>
<td>5</td>
<td>8</td>
<td>25</td>
<td>0.07</td>
<td>23</td>
</tr>
<tr>
<td>t481</td>
<td>16</td>
<td>1</td>
<td>481</td>
<td>2.86</td>
<td>481</td>
</tr>
<tr>
<td>table3</td>
<td>14</td>
<td>14</td>
<td>175</td>
<td>3.81</td>
<td>175</td>
</tr>
<tr>
<td>table5</td>
<td>17</td>
<td>15</td>
<td>158</td>
<td>2.52</td>
<td>158</td>
</tr>
<tr>
<td>vg2</td>
<td>25</td>
<td>8</td>
<td>110</td>
<td>0.58</td>
<td>110</td>
</tr>
<tr>
<td>xor5</td>
<td>5</td>
<td>1</td>
<td>16</td>
<td>0.02</td>
<td>16</td>
</tr>
</tbody>
</table>
marks, reported in [51]. Unfortunately, the running times are not given in [51], so we cannot make a comparison with AOXMIN in terms of time. Also, we cannot compare the performance of AOXMIN to the performance of the multi-level AND-OR-XOR minimizer from [8], because their results are given in terms of the literal count (i.e., the number of complemented and uncomplemented variables occurring in each product), and not in terms of the number of product-terms, as in our algorithm.

Columns 2 and 3 give the number of inputs $n$ and the number of outputs $m$ of the logic circuits implementing the benchmark functions. The last column $N_{\text{iter}}$ refers to the number of iterations performed by our algorithm. For each function, we tried 1, 10, and 20 iterations and we show the lowest number of iterations for achieving the best result.

In terms of the number of product-terms, for 15 of the 25 benchmarks AOXMIN gives a smaller result than Espresso without output phase optimization. On average, the number of product-terms obtained by Espresso is 2.13 times larger than the number of product-terms obtained by our algorithm. However, in terms of time, Espresso without output phase optimization is on average 5.27 times faster than AOXMIN.

When compared to Espresso with output phase optimization (using -Dopo option), for 10 of the 25 benchmarks AOXMIN obtains a result with fewer product-terms. On average the number of product-terms obtained by our algorithm is 1.11 times larger, but it is 1.17 times faster.

Compared to the AND-OR-XOR minimizer from [51], AOXMIN generates almost the same results in terms of number of product-terms, with the exception of $t481$ function, for which our algorithm considerably reduces the number of product-terms. We found out that, using AOXMIN, the number of product-terms in $t481$ can be further reduced if more iterations of the algorithm are performed (see Table 4.2). For 50 iterations the number of product-terms for $t481$ is 18, but the program needs 24.4 min to compute it. The resulting AND-OR expansions for the functions $g1\_best$ and
CHAPTER 4. AND-OR-XOR MINIMIZATION OF BOOLEAN FUNCTIONS

$g_{1\text{.best}}$ are:

$$g_{1\text{.best}} = x'_1 x'_2 x'_3 x'_4 + x'_5 x'_7 x'_8 + x_5 x_7 x_8 + x'_3 x'_6 x'_7 + x_1 x'_3 + x_1 x_4 + x'_2 x'_3 + x'_2 x_4$$

$g_{2\text{.best}}$ are:

$$g_{2\text{.best}} = x'_1 x'_3 x'_4 x'_5 x'_6 + x'_1 x'_5 x'_6 + x'_9 x_1 x'_12 + x'_9 x_1 x'_12 + x'_9 x_10 x'_11 + x'_14 x'_16 + x_1 x'_16$$

Notice that the functions have disjoint variable sets and interesting symmetry properties.

Table 4.2: AOXMIN results for $t481$.

<table>
<thead>
<tr>
<th>product-terms</th>
<th>time/sec</th>
<th>$N_{\text{iter}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>228</td>
<td>47.4</td>
<td>1</td>
</tr>
<tr>
<td>134</td>
<td>295.7</td>
<td>10</td>
</tr>
<tr>
<td>113</td>
<td>557.2</td>
<td>20</td>
</tr>
<tr>
<td>18</td>
<td>1461.5</td>
<td>50</td>
</tr>
</tbody>
</table>

The experimental results indicate that our algorithm works quite well for functions with embedded XOR logic (like $5xpl.clip.rd53.rd73.sao2.t481$). On the other hand, for benchmarks without embedded XOR logic, like $\text{misex1, misex2}$ and $\text{ex1010}$, introducing XOR does not bring any advantage. The "hardest" function for our program is $\text{ex1010}$, for which the program is unable to find a result with less than 725 product-terms for up to 100 iterations.

As can be seen, in general, the time required for AOXMIN to find a solution is quite reasonable, especially taking into account the complexity of the problem. The most time-consuming are the calls to Espresso. We hope in the future to improve the time-performance of AOXMIN by making it an integrated program. Presently the problem is treated in several stages, by first computing minimal AND-OR-XOR expansions for each output functions separately, and then finding common subterms for the resulting functions.
CHAPTER 4. AND-OR-XOR MINIMIZATION OF BOOLEAN FUNCTIONS

4.6 Conclusion

This chapter presents a new heuristic algorithm for minimizing AND-OR-XOR expansions of incompletely specified Boolean functions with a worst-case time complexity \( O(N^3) \), where \( N = 2^n \) and \( n \) is the number of variables in the function. As with any heuristic algorithm, ours does not guarantee that a minimal solution is found, but usually obtains a nearly-minimal one. However, as the experimental results show, the algorithm has a satisfactory performance for several well-known benchmark functions. We also prove an upper bound of \( 5 \cdot 2^{n-4} \) on the number of product-terms in the AND-OR-XOR expansion.

A drawback of our algorithm is that it is only capable of finding the AND-OR-XOR expansions which obey the constraint that each product-term from a minimal AND-OR expansion of the function is entirely contained in either \( g_1 \) or \( g_2 \), which is, of course, not necessary in general. Since the minimal AND-OR expansion of a function is often not unique, the performance of our algorithm could be improved by trying several minimal AND-OR expansions as starting points. But this still does not guarantee that the resulting solution is a minimal one.

Two major facets of our method require further research. First, as noted above, we use a very simple approach to handle multiple-output circuits. Functions describing the constituent single-output circuits are treated separately until the final minimization step. As in conventional two-level minimization, we expect to achieve much better results if a method can be found to treat the functions simultaneously throughout the complete process. The difficulty is how to extend the notion of cube chains to the multiple-output case.

The second area for further work is the selection of partitionings. At present, we select these pseudo-randomly. We need to examine which functional properties can be used to guide the choice of partitionings and also to investigate which search techniques are applicable to this problem.
Chapter 5

Test Generation for Multiple-Valued Circuits

This chapter introduces a new multiple-valued discrete difference, which we call full sensitivity, and shows its application to generating tests for multiple-valued logic circuits. Full sensitivity is also used in Chapter 6 to define a class of multiple-valued functions $\Sigma$, studied there with respect to decomposition.

Section 5.1 gives a background on test generation for logic circuits. Section 5.2 shows how Boolean difference can be used for generating tests and describes previous work on generalization of Boolean difference to the multiple-valued case. In section 5.3, full sensitivity is introduced. An algorithm for determining the full sensitivity is given in section 5.4. The application of full sensitivity to test generation is discussed in section 5.5. In section 5.6 a lower bound on the number of functions, fully sensitive to their variables is derived.

The material of this section is based on [18], [19] and [17].

5.1 Test generation for logic circuits

This section briefly describes the basic notions from test generation, necessary for this chapter and Chapter 7.

A fault of an electrical circuit is a physical defect of one or more components, which can cause the circuit to malfunction. Many physical faults in electrical circuits
can be modeled by a *stuck-at fault* logic model. In this kind of fault-model it is assumed that any physical fault (such as, for example, short or open diode, broken wire between gates, etc.) can be modeled by a number of lines in the corresponding logic circuit permanently fixed at a logic level 0, 1, ..., or $m - 1$. In this chapter, we use the *single stuck-at fault* logic model, assuming that a single line in the circuit is fixed at a logic level 0, 1, ..., or $m - 1$.

Any $n$-tuple $(a_1, \ldots, a_n) \in M^n$ of values of the input variables $(x_1, \ldots, x_n)$ is called *test* for a fault $\alpha$ if and only if

$$f(a_1, \ldots, a_n) \neq f^\alpha(a_1, \ldots, a_n)$$

where $f^\alpha(a_1, \ldots, a_n)$ denotes the function describing the circuit in the presence of the fault $\alpha$ [58]. Such a test $(a_1, \ldots, a_n)$ is said to detect the fault $\alpha$.

In the traditional methods for fault detection, tests are applied to the circuit under test and the output responses are verified one by one. Any discrepancy detects a fault. A set $T$ of input vectors is called a *test set* for some set $F$ of faults if the observation of the corresponding outputs allows the detection of every fault from $F$ in the circuit.

If the circuit is not redundant, the set of $m^n$ possible input vectors is a test set of the circuit. However, if the circuit implementation is known, it is possible to construct tests sets having less than $m^n$ elements. One of the objectives of testing is to construct minimal test sets. A test set is called *minimal* if it ceases to be a test provided any single elements is deleted. Usually, the search for minimal tests is usually carried out in two distinct parts:

1. For each of the possible faults, generate the corresponding tests.

2. Find a *minimal cover* of the faults by a set of test vectors.

In the next section we show how Boolean difference can be used for generating tests for a logic circuit.
5.2 Boolean difference for test generation

Boolean difference is the basis of one of the well-known algorithms for generating tests for Boolean logic circuits. It makes use of an algebraic description of the fault free circuit and manipulates this description to generate tests for stuck-at faults. If a combinational circuit realizes the function \( f(x_1, \ldots, x_n) \), then the Boolean difference \( \frac{df}{dx_i} \) represents all conditions under which the value of \( f \) is sensitive to the value of \( x_i \), i.e., when a change in the value of \( x_i \) from one logic value to another causes a change in the value of the function \( f \). Thus \( x_i \cdot \frac{df}{dx_i} \) represents the set of all tests for the fault \( x_i \) stuck-at-0, since \( x_i \) applies the opposite logic value on the faulty input and the term \( \frac{df}{dx_i} \) ensures that this erroneous signal affects the value of \( f \). Similarly, the set of all tests which detect \( x_i \) stuck-at-1 is defined by the expression \( x'_i \cdot \frac{df}{dx_i} \).

The notion of Boolean difference is well-known and is defined by:

\[
\frac{df(x_i)}{dx_i} = f(x_i^0) \div f(x_i^1)
\]

for \( i \in \mathcal{V} \). Boolean difference reflects how the value of \( f \) is affected when the value of the variable \( x_i \) changes from 0 to 1. While in a Boolean algebra with carrier set \( B = \{0, 1\} \) the choice of a change \( dx_i \) to be equal to 1 is unique and intuitively clear, in a multiple-valued algebra with carrier set \( M = \{0, 1, \ldots, m-1\} \) the choice of \( dx_i \) can be treated in different ways. Several generalizations of Boolean difference to the multiple-valued case exist ([26], [35], [60]), all based on different interpretations of \( dx_i \). These generalizations have one common feature - they allow the change of \( x_i \) to be any possible change from a logic value \( a \) to a logic value \( b \). \( a, b \in M \). Any of the algorithms from [26], [35] and [60] allows the generation of tests for all detectable stuck-at faults in a circuit, but their complexity is very high in that many differences (at least \( m \)) have to be calculated to generate tests for all the stuck-at faults on just one line in the circuit.

In the approach proposed in this chapter an attempt is made to exploit the structure of the value change \( dx_i \) more deeply, in order to avoid unnecessary calculations.
CHAPTER 5. TEST GENERATION FOR MULTIPLE-VALUED CIRCUITS

during test generation. We define a generalization of Boolean difference, called full sensitivity, not as a quantitative measure of the exact change of $f(x)$ with respect to a given change of $x_i$, but as a qualitative measure, reflecting whether $f(x)$ changes anytime the value of $x_i$ is changed. As a result, to test a line for all single stuck-at faults, only one difference per line has to be calculated (as in the two-valued case), instead of at least $m$ differences as in algorithms from [26], [35] and [60].

A penalty for this simplicity is that tests for some detectable faults cannot be generated using full sensitivity. However, we show in this chapter that, for practical values of $m$ and $n$, the percentage of such cases is extremely small.

5.3 Definition of full sensitivity

Boolean difference is defined through an XOR operation $\oplus$. An XOR detects whether its two arguments are distinct or not. Similarly, we define full sensitivity through a generalization of the XOR operation which detects whether its $m$ arguments are pairwise distinct or not. We call this new operation mutual exclusion.

**Definition 5.1** Mutual exclusion is the $m$-ary operation defined by

$$mx(x_0, x_1, \ldots, x_{m-1}) := \begin{cases} 
    m - 1 & \text{if } x_i \neq x_j \text{ for all } i \neq j \\
    0 & \text{otherwise}
\end{cases}$$

where $i, j \in M$ and $x_0, x_1, \ldots, x_{m-1}$ are variables ranging over $M$.

Such operations (or functions) which take values in $\{0, (m - 1)\}$ only are called decisive. In terms of the mutual exclusion, full sensitivity can be expressed as follows.

**Definition 5.2** The full sensitivity of $f(x)$ with respect to $x_i$ is defined as

$$\frac{F S f(x)}{F S x_i} = mx[f(x^0_i), f(x^1_i), \ldots, f(x^{m-1}_i)]$$

where $i \in N$. 
If $\frac{FSf(x)}{FSx_i} \neq 0$, when the algebraic expression of the full sensitivity indicates for which fixed values of the variables $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots x_n$ the value of the function $f(x)$ changes anytime the value of $x_i$ changes. If $\frac{FSf(x)}{FSx_i} \neq 0$, then we say that $f(x)$ is fully sensitive to $x_i$.

If $m = 2$, then

$$\frac{FSf(x)}{FSx_i} = mx[f(x_i^0), f(x_i^1)] = f(x_i^0) \oplus f(x_i^1).$$

i.e. for the two-valued case full sensitivity reduces to a Boolean difference.

**Example 5.3.** Consider the 3-valued function of two variables, shown in Figure 5.1a.

The full sensitivities $\frac{FSf(x)}{FSx_i}, i \in \{1, 2\}$, may be calculated from the Karnaugh map of the function in accordance with Definition 5.2. Thus, $\frac{FSf(x)}{FSx_i}$ is different from constant zero if, for some fixed value $a \in M$ of the other variable, the subfunction $x_i \mapsto f(x_i, a)$ is bijective (a permutation of $M$). For $f(x_1, x_2)$ in Figure 5.1a, the subfunctions $x_1 \mapsto f(x_1, 0)$, $x_1 \mapsto f(x_1, 1)$ and $x_2 \mapsto f(2, x_2)$ are such bijections. So,

$$\frac{FSf(x)}{FSx_1} = x_2 + x_2$$

and

$$\frac{FSf(x)}{FSx_2} = x_1.$$ 

as shown in Figure 5.1 (b, c).

In the next section we develop an algorithm for computing the full sensitivity.

### 5.4 Calculation of full sensitivity.

Definition 5.2 provides an easy way of finding the full sensitivity if a function is given by its Karnaugh map. In this section we present an algorithm allowing us to calculate $\frac{FSf(x)}{FSx_i}$ from its algebraic expansion. To develop such an algorithm, we
need to express mutual exclusion and full sensitivity in terms of some functionally complete set of functions. Here, we use the operations of chain-based Post algebra (Definition 2.1), but other choices are also possible.

Let $\Psi$ be the set of all permutations over $M$, i.e.

$$\Psi := \{(a_0, a_1, \ldots, a_{m-1}) \mid a_i \in M \land i \in M \land mx(a_0, a_1, \ldots, a_{m-1}) = (m - 1)\}.$$

Then, the algebraic expression for mutual exclusion in terms of the operation $\text{MAX}$, $\text{MIN}$, and literals is described by Property 5.4. The sign $\sum$ used in this property and elsewhere in the chapter denotes $\text{MAX}$, and the sign $\prod$ denotes $\text{MIN}$.

**Property 5.4** Let $x_0, x_1, \ldots, x_{m-1}$ be variables ranging over $M$.

$$mx(x_0, x_1, \ldots, x_{m-1}) = \sum_{(a_0, \ldots, a_{m-1}) \in \Psi} a_0 x_0 \cdot \ldots \cdot x_{m-1}$$

$$= \sum_{(a_0, \ldots, a_{m-1}) \in \Psi} \prod_{i \in M} x_i$$
CHAPTER 5. TEST GENERATION FOR MULTIPLE-VALUED CIRCUITS

This property can easily be proved using Definition 5.1 and the definition of literals.

For \( m = 3 \):

\[
m(x_0, x_1, x_2) = x_0 x_1 x_2 + x_0 x_1 x_2 + x_0 x_1 x_2 + x_0 x_1 x_2 + x_0 x_1 x_2 + x_0 x_1 x_2.
\]

Every \( m \)-valued function of \( n \)-variables can be "partitioned" into \( m \) functions of \((n - 1) \) variables using the generalized Shannon decomposition (2.1) as follows:

\[
f(x) = f(x_0) + f(x_1) + \ldots + f(x_{m-1})
\]

On the other hand by we can write an \( m \)-valued function in an expansion consisting of \( m \) decisive functions \( f(x) \). \( k \in M \), where \( f(x) \) denotes a literal of the function \( f(x) \). namely

\[
f(x) = \sum_{k \in M} k \cdot f(x)
\]

In Table 5.1 we show these two possible representations for the function from Example 5.3.

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( f(x_1, x_2) )</th>
<th>( \delta_0 f(x_0) )</th>
<th>( \delta_1 f(x_1) )</th>
<th>( \delta_2 f(x_2) )</th>
<th>( \delta_0 f(x_1, x_2) )</th>
<th>( \delta_1 f(x_1, x_2) )</th>
<th>( \delta_2 f(x_1, x_2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

When we use Definition 5.2 to find full sensitivities, we utilize the decomposition from (5.1). But the equation (5.2) provides us with more a convenient way for cal-
The next property shows that full sensitivities can be calculated as the MIN of \( m \) functions.

For notational simplicity, we use \( \hat{f}(x_i^o) \) to denote \( \sum_{j \in M} \hat{f}(x_i^j) \).

**Property 5.5**

\[
\frac{FSf(x_i)}{FSx_i} = f(x_i^0) \cdot f(x_i^1) \cdot \ldots \cdot f(x_i^{m-1}) = \prod_{k \in M} \hat{f}(x_i^k).
\]

**Proof:**

\[
\frac{FSf(x_i)}{FSx_i} = mx(f(x_i^0), f(x_i^1), \ldots, f(x_i^{m-1})) \quad \{\text{Definition 5.2}\}
\]
\[
= \sum_{a_0, \ldots, a_{m-1} \in \Psi} \prod_{k \in M} \hat{f}(x_i^k) \quad \{\text{Property 5.4}\}
\]
\[
= \sum_{a_0, \ldots, a_{m-1} \in \Psi} \prod_{k \in M} \hat{f}(x_i^{a_k}) \quad \{\text{Commutativity of MIN}\}
\]
\[
= \sum_{a_0, \ldots, a_{m-1} \in \Psi} \prod_{k \in M} \hat{f}(x_i^{a_k}) \quad \{a \neq b \rightarrow \hat{f}(x_i^a) \cdot \hat{f}(x_i^b) = 0\}
\]
\[
= \prod_{k \in M} \sum_{j \in \Psi} \hat{f}(x_i^j) \quad \{\text{Distrib. of MAX over MIN}\}
\]
\[
= \prod_{k \in M} \hat{f}(x_i^k) \quad \{\text{Def. of } \hat{f}(x_i^k)\}
\]

\( \square \)

There is a simple connection between the functions \( \hat{f}(x_i^o) \) and \( \hat{f}(x_i) \). Expanding \( \hat{f}(x_i) \) using (5.2) we have:

\[
\hat{f}(x_i) = x_i \hat{f}(x_i^0) + \hat{x_i} \hat{f}(x_i^0) + \ldots + \hat{x_i}^{m-1} \hat{f}(x_i^{m-1}).
\]

Comparing this expression with the definition of \( \hat{f}(x_i^o) \) we can see that the functions \( \hat{f}(x_i^o) \) can be obtained from \( \hat{f}(x_i) \) by replacing symbolically all occurrences of the literals \( \hat{x}_i \) in the expression of \( \hat{f}(x_i) \) by \( (m - 1) \). \( i \in N, j, k \in M \).

So, an algorithm for finding \( \frac{FSf(x_i)}{FSx_i} \) for a given \( i \in N \) has the following steps:

1. Express the function \( f(x) \) as (5.2), expanding the functions \( \hat{f}(x_i) \), \( k \in M \). in the MIN-MAX canonical form (Theorem 2.2).
2. In each \( \hat{f}(x) \), replace symbolically all occurrences of the literal \( \hat{x}_j \) by \( m - 1 \). (\( j, k \in M \)):

3. Take MIN of the functions obtained in step 2. By Property 5.5, the result is the full sensitivity \( \frac{FS(f)}{FSx_i} \).

Example 5.6. We compute \( \frac{FS(f)}{FSx_i} \) for the function from Example 5.3 using the algorithm described above.

1. The function has the following (5.2) form:
   \[
   f(x) = 0 \cdot x_1 x_2 + 0 \cdot x_1 \hat{x}_2 + 2 \cdot \hat{x}_1 \hat{x}_2 + 1 \cdot \hat{x}_1 \hat{x}_2 + 1(\hat{x}_1 x_2 + \hat{x}_1 \hat{x}_2 + 2\hat{x}_1 \hat{x}_2) + 2(\hat{x}_1 \hat{x}_2 + \hat{x}_1 \hat{x}_2).
   \]

Here, the functions \( \hat{f}(x), \hat{f}(x) \) and \( \hat{f}(x) \) in the MIN-MAX canonical form are:

\[
\hat{f}(x) = \begin{cases} 0 & x_1 x_2 + x_1 \hat{x}_2 + 2 \cdot \hat{x}_1 \hat{x}_2 + 1 \cdot \hat{x}_1 \hat{x}_2 \\ 1 & x_1 x_2 + 1 \cdot \hat{x}_1 \hat{x}_2 + 2 \cdot \hat{x}_1 \hat{x}_2 \\ 2 & x_1 x_2 + 2 \cdot \hat{x}_1 \hat{x}_2 \end{cases}
\]

2. Replacing all occurrences of literals \( \hat{x}_1, \hat{x}_1 \) and \( \hat{x}_1 \) by \( m - 1 = 2 \) we have
   \[
   \hat{f}(x) = 2 \cdot x_2 + 2 \cdot \hat{x}_2 + 2 \cdot \hat{x}_2 + 2 \cdot \hat{x}_2 = 2
   \]

3. Applying Property 5.5 we get
   \[
   \frac{FS(f)}{FSx_i} = 2 \cdot 2 \cdot (\hat{x}_2 + \hat{x}_2) = \hat{x}_2 + \hat{x}_2.
   \]

Step 1 of the algorithm requires expanding the function in the MIN-MAX canonical form with no simplification used (i.e. \( m^n \) terms for an \( m \)-valued \( n \)-variable function). Therefore, the algorithm becomes infeasible for large values of \( m \) and \( n \). Further work needs to be done to find a more efficient algorithm for computing full sensitivity, using a more compact representation of functions, for example multiple-valued decision diagrams [40].

Since Boolean difference is used for generating tests for Boolean logic circuits and full sensitivity is a generalization of Boolean difference, it is natural to expect that
full sensitivity can be used for generation of tests for multiple-valued logic circuits. This question is considered in the next section.

5.5 Full sensitivity in test generation.

In this section we describe how full sensitivity can be used to generate tests for single stuck-at faults on a given line in a logic circuit.

5.5.1 Test generation for primary inputs

Let \( x_i \) be the variable corresponding to the \( i_{th} \) primary input, \( i \in N \) in a logic circuit realizing a multiple-valued function \( f(x_1, \ldots, x_n) \). The full sensitivity \( \frac{FS(f(x_i \ldots x_n))}{FS_x} \) represents all conditions (associated with variables in \( x_1, \ldots, x_n \), excluding \( x_i \)) under which the value of \( f \) is fully sensitive to the value of \( x_i \), i.e., when any change in the value of \( x_i \) from one logic value to another causes a change in the value of the function \( f \). Thus \( x_i \cdot \frac{FS(f(x_i \ldots x_n))}{FS_x} \) represents a set of tests for all the faults \( x_i \) stuck-at-\( \overline{k} \), where \( \overline{k} \) denotes any value but \( k \), since \( x_i \) applies the logic value \( k \) different from \( \overline{k} \), on the faulty input and the factor \( \frac{FS(f(x_i \ldots x_n))}{FS_x} \) ensures that this erroneous signal affects the value of \( f \). Similarly, a set of tests which detect the remaining \( x_i \) stuck-at- \( k \) fault is defined by the expression \( x_i \cdot \frac{FS(f(x_i \ldots x_n))}{FS_x} \) for any \( j \in M - \{k\} \). These two sets of tests, if different from the empty set, cover all detectable stuck-at faults on the \( i_{th} \) primary input.

However, if the full sensitivity of \( f(x_1, \ldots, x_n) \) with respect to the variable \( x_i \) is a constant zero function, then the output of the circuit cannot be made fully sensitive to the primary input \( i \), and tests for single stuck-at faults on this primary input cannot be generated using full sensitivity.

For example, consider the circuit shown in Figure 5.2. Let \( m = 3 \) and let the inverter gate be defined to realize the complement \( x' \). Then the circuit realizes the function \( z = f(x_1, x_2) = 1(x_1 + x_1^2) x_2 \). In this case \( \frac{FS_z}{FS_{x_1}} = 0 \) and \( \frac{FS_z}{FS_{x_2}} = 0 \) so that tests for single stuck-at faults on the primary inputs cannot be generated using full sensitivity.
sensitivity. However, such tests can be derived by using one of the algorithms from [26], [35] or [60]. For example, the input combination \((x_1, x_2) = (0, 1)\) is a test for \(x_1\) stuck-at-2 and for \(x_1\) stuck-at-1 faults.

\[
\text{Figure 5.2: Example circuit.}
\]

So, in order for full sensitivity to allow us to generate tests for all detectable single stuck-at faults on all primary inputs, one condition has to be met: for each \(x_i\) there has to exist an assignment \((a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)\) for the other input variables \((x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)\) for which the full sensitivity of \(f(x_1, \ldots, x_n)\) with respect to the variable \(x_i\) takes a logic value different from zero, i.e.

\[
\left. \frac{FSf(x_1, \ldots, x_n)}{FSx_i} \right|_{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)} \neq 0.
\]

where \(a_j \in M. \ j \in \{1, \ldots, i-1, i+1, \ldots, n\}\).

In section 5.6 we investigate in how many cases this condition is met. We can see it does not depend on a particular circuit realization and can be checked on the functional level. All functions which are fully sensitive to all their variables, i.e. for which:

\[
\frac{FSf(x_1, \ldots, x_n)}{FSx_i} \neq 0 \text{ for all } x_i. \ i \in N.
\]

satisfy this condition.

5.5.2 Test generation for internal lines

Let \(x_p\) be a variable, corresponding to some internal line \(p\) in the circuit (Figure 5.3).

The value of \(x_p\) is some function of the input variables, i.e. \(x_p = h(x_1, \ldots, x_n)\). Let us denote with \(g(x_1, \ldots, x_n, x_p)\) the function, modeling the circuit after "cutting" line \(p\) and considering \(x_p\) to be a new input variable.
If the full sensitivity of $g(x_1, \ldots, x_n, x_p)$ with respect to the variable $x_p$ is not a constant zero function, then the output of the circuit can be made, by the application of some input vector $(a_1, \ldots, a_n)$, fully sensitive to line $p$. Suppose that under application of $(a_1, \ldots, a_n)$ the logic value on the line $p$ is $k$, i.e.

$$x_p = h(a_1, \ldots, a_n) = k, \quad k \in M.$$  

Then the input vector $(a_1, \ldots, a_n)$ is a test for all single stuck-at-$\overline{k}$ faults on line $p$, where $\overline{k}$ denotes any logic value but the logic value $k$. To generate tests for the remaining single stuck-at-$k$ fault, there has to exist a second assignment for input variables $(b_1, \ldots, b_n) \neq (a_1, \ldots, a_n)$, which makes the output of the circuit fully sensitive to the line $p$ and applies a logic value different from $k$ on the faulty line $p$.

For example, for the circuit shown in Figure 5.2, the output $z$ can be made fully sensitive to the internal line $x_3$ by application of the input vector $x_1 = 2$, $x_2 = 0$. Under the application of this input vector the logic value on the line $x_3$ is 0. So, the input vector $(x_1, x_2) = (2, 0)$ is a test for all single stuck-at-$j$ faults on the line $x_3$, $j \in \{1, 2\}$. Since there exists no other assignment of input variables, which makes the output $z$ fully sensitive to the line $x_3$ and applies a logic value different from 0 on line $x_3$, tests for $x_3$ stuck-at-0 cannot be generated using full sensitivity.

Summarizing, in order for full sensitivity to allow us to generate tests for all detectable single stuck-at faults on an internal line $p$, two conditions have to be met:
1. There exists an assignment for input variables \((a_1, \ldots, a_n)\) for which the full sensitivity of \(g(x_1, \ldots, x_n, x_p)\) with respect to the variable \(x_p\) takes a logic value different from zero, i.e.:

\[
\left. \frac{FSg(x_1, \ldots, x_n, x_p)}{FSx_p} \right|_{(a_1, \ldots, a_n)} \neq 0.
\]

where \(a_j \in M\). \(j \in N\).

This condition guarantees generating tests for \((m - 1)\) single stuck-at faults on internal line \(p\).

2. There exists a distinct assignment for input variables \((b_1, \ldots, b_n) \neq (a_1, \ldots, a_n)\) for which the full sensitivity of \(g(x_1, \ldots, x_n, x_p)\) with respect to the variable \(x_p\) takes a logic value different from zero, and for which the logic value on the line \(p\) is different from the logic value on line \(p\) under the application of input vector \((a_1, \ldots, a_n)\), i.e.:

\[
h(a_1, \ldots, a_n) \neq h(b_1, \ldots, b_n).
\]

\(a_j, b_j \in M\). \(j \in N\).

This condition guarantees generating tests for the last single stuck-at fault on internal line \(p\).

The next section establishes the number of \(m\)-valued \(n\)-variable functions which are fully sensitive to some distinguished variable. as compared to all \(m\)-valued \(n\)-variable functions. This gives us an estimate of how often the first condition is met.

The second condition cannot be analyzed on the functional level, because it depends on a particular circuit realization.
5.6 **Total number of m-valued functions fully sensitive to all their variables**

In this section we derive a lower bound on the number of m-valued n-variable functions that are fully sensitive to all their variables.

The results we are presenting are based on the following intuition. Consider a set $U$ (Figure 5.4) and $n$ subsets $A_1, A_2, \ldots, A_n$ of $U$, which have the same cardinality. i.e. $|A_1| = |A_2| = \ldots = |A_n|$. The set $U$ represents the set of all $m$-valued functions of $n$-variables. Each of $A_i$, $i \in N$, represents the set of functions which are fully sensitive to the variable $x_i$.

We would like to know how many functions are fully sensitive to all their variables $x_1, x_2, \ldots, x_n$, i.e. what is the cardinality of the intersection of the subsets $A_1, A_2, \ldots, A_n$. It seems very hard to obtain an exact formula for the size of this intersection. Much simpler is to find the size of one subset $|A_i| = a$, $i \in N$, and then, using the relationship

$$|A_1 \cap A_2 \cap \ldots \cap A_n| \geq |U| - n \cdot (|U| - a).$$

(5.3)

to derive a lower bound on the cardinality of the set $A_1 \cap A_2 \cap \ldots \cap A_n$.

Let $N_{m,n}$ denote the number of all $m$-valued functions of $n$-variables, i.e.
CHAPTER 5. TEST GENERATION FOR MULTIPLE-VALUED CIRCUITS

\[ N_{m,n} = m^{(m^n)}. \]

Notice that \( N_{m,n} \) also includes all degenerate \( m \)-valued \( n \)-variable functions, i.e. functions which do not depend upon all \( n \) variables to determine function output (one or more variables are redundant) \([30]\). The following Lemma gives the number of all \( m \)-valued \( n \)-variable functions which are fully sensitive to one selected variable \( x_i \).

**Lemma 5.7** The number of \( m \)-valued \( n \)-variable functions which are fully sensitive to a variable \( x_i \), where \( i \) is some fixed value in the set \( \{1, 2, \ldots, n\} \), is:

\[ N_{FS_{x_i}} = N_{m,n} - (m^m - m!)^{(m^n - 1)}. \]

**Proof:** We prove that the number of \( m \)-valued \( n \)-variable functions which are not fully sensitive to a variable \( x_i \) is:

\[ (m^m - m!)^{(m^n - 1)}. \]

Let \([f_0, f_1, \ldots, f_{m^n-1}]\) denote the values from the truth table of an \( m \)-valued \( n \)-variable function \( f \). Without loss of generality we consider the case when \( x_i \) is the least significant variable, i.e. \( x_n \). It follows from the definition of full sensitivity (Definition 5.2) that a function \( f \) is not fully sensitive to the variable \( x_n \) if and only if, for all \( j \in \{0, 1, \ldots, m^n - 1\} \),

\[ mx(f_{m_j}, f_{m_j+1}, \ldots, f_{m_j+m-1}) = 0. \]

The mutual exclusion operation equals zero when its \( m \) arguments are not pairwise distinct. So, the number of \( m \)-tuples, satisfying

\[ mx(f_{m_j}, f_{m_j+1}, \ldots, f_{m_j+m-1}) = 0 \]

for a fixed \( j \) is:

\[ m^m - m! \]

where \( m^m \) is the number of all possible combinations of \( m \) digits over an \( m \)-tuple, and \( m! \) is the number of all possible permutations of \( m \) digits.
CHAPTER 5. TEST GENERATION FOR MULTIPLE-VALUED CIRCUITS 68

Since the condition $m x(f_{mj}, f_{mj+1}, \ldots, f_{mj+m-1}) = 0$ has to be met for all $j \in \{0, 1, \ldots, m^{n-1} - 1\}$, the number of $m$-valued $n$-variable functions that are not fully sensitive to the variable $x_n$ is:

$$(m^m - m!)(m^{n-1}).$$

□

Having proved the lemma, we now show that the percentage of functions which are not fully sensitive to a distinguished variable tends to zero as $n$ increases. Consider the fraction

$$\frac{N_{FS, to-x_i}}{N_{m,n}} = 1 - \frac{(m^m - m!)(m^{n-1})}{N_{m,n}} = 1 - e(m, n)$$

where $e(m, n) := \frac{(m^m - m!)(m^{n-1})}{N_{m,n}}$.

The fraction $e(m, n)$ characterizes how many of the functions are not fully sensitive

![Figure 5.5: Plots for $e(m, n)$ as a function of $n$ for fixed $m = 3, 5$ and 10.](image)
to a variable $x_i$ as compared to all $m$-valued $n$-variable functions. For example, $e(m, n) = 0.1$ means that 10% of all $m$-valued $n$-variable functions are not fully sensitive to one selected variable $x_i$. By applying the Stirling formula for a factorial [7], the expression for $e(m, n)$ can be simplified as:

$$e(m, n) = \left(1 - \frac{\sqrt{2\pi m}}{e^m}\right)^{m^{n-1}}.$$ 

It is easy to show that for any fixed $m > 1$

$$\lim_{n \to \infty} e(m, n) = 0.$$ 

Consequently, as $n$ increases, the fraction of the functions not fully sensitive to a variable $x_i$ declines to zero. It can be seen from the plots for $e(m, n)$ as a function of $n$ for fixed $m = 3, 5$ and 10 (Figure 5.6) that it declines rapidly. For $n = 5$ the values of $e(3, 5), e(5, 5)$ and $e(10, 5)$ are almost zero.

If $N_{FS_{to\_all}}$ is the number of $m$-valued functions fully sensitive to all their variables $x_1, x_2, \ldots, x_n$, then we define $e^*(m, n)$ by

$$e^*(m, n) := 1 - \frac{N_{FS_{to\_all}}}{N_{m,n}}.$$ 

Hence, $e^*(m, n)$ characterizes how many of the $m$-valued functions are not fully sensitive to all their variables $x_1, x_2, \ldots, x_n$ as compared to all $m$-valued $n$-variable functions. As we mentioned earlier, it seems very hard to find a closed formula for $N_{FS_{to\_all}}$. But using the relationship from (5.3) it is easy to establish that

$$e^*(m, n) \leq n \cdot e(m, n).$$

Recall, that in terms of the considerations from the beginning of this section $N_{m,n} = |U|$, $N_{FS_{to\_x_i}} = |A_i| = a$ and $N_{FS_{to\_all}} = |A_1 \cap A_2 \cap \ldots \cap A_n|$. Thus, (5.3) can be rewritten as:

$$N_{FS_{to\_all}} \geq N_{m,n} - n \cdot (N_{m,n} - N_{FS_{to\_x_i}}).$$ (5.4)
Theorem 5.8 Let $e^*(m, n)$ be defined as above. The following inequality holds:

$$e^*(m, n) \leq n \cdot (1 - \frac{\sqrt{2\pi m}}{e^m})^{(m^{n-1})}.$$ 

**Proof:** By dividing both sides of (2) by $N_{m,n}$ we obtain:

$$\frac{N_{FS,to\text{-}all}}{N_{m,n}} \geq 1 - n \cdot \left(1 - \frac{N_{FS,to\text{-}all}}{N_{m,n}}\right).$$

Since $\frac{N_{FS,to\text{-}all}}{N_{m,n}} = 1 - e(m, n)$ and $e^*(m, n) := 1 - \frac{N_{FS,to\text{-}all}}{N_{m,n}}$, we have

$$e^*(m, n) \leq n \cdot e(m, n) = n \cdot \left(1 - \frac{\sqrt{2\pi m}}{e^m}\right)^{(m^{n-1})}.$$

\[\square\]

Figure 5.6: Plots for $e(3, n)$ and $e^*(3, n)$ as functions of $n$ for $m = 3$.

It follows from the Theorem 5.8, that for values of $n$ for which $e(m, n)$ is extremely small, the value of $e^*(m, n)$ is extremely small, too. Figure 5.6 illustrates this for the case $m = 3$. For practical values of $n$ and $m$, the fraction of $m$-valued functions not
fully sensitive to all their variables is extremely small. For example, for \( n > 5 \) when \( 3 \leq m \leq 10 \), the percentage of \( m \)-valued \( n \)-variable functions not fully sensitive to all their variables is less than \( 1.38 \times 10^{-13} \% \). Therefore for large \( n \) and \( m \) almost all \( m \)-valued \( n \)-variables functions are fully sensitive to all their variables.

5.7 Conclusion

In this chapter we introduce a new multiple-valued discrete difference and investigate its usefulness to test generation for multiple-valued logic circuits. For this purpose, a closed formula for the number of functions which are fully sensitive to one of the variables is proved. Using this formula, a lower bound is derived for the number of \( m \)-valued \( n \)-variable functions that are fully sensitive to all their variables. From the result, we can make the following conclusions in terms of test generation.

Since for practical values of \( m \) and \( n \) (\( 3 \leq m \leq 10 \), \( n > 5 \)) the percentage of functions fully sensitive to all their variables is very high, it appears probable that tests for all single stuck-at faults on primary inputs can be generated using full sensitivity for almost all instances.

For an internal line \( p \), if \( g(x_1, \ldots, x_n, x_p) \) is fully sensitive to \( x_p \), then tests for at least \( m - 1 \) single stuck-at faults on that line can be generated using full sensitivity. Notice, however, that since the function \( g(x_1, \ldots, x_n, x_p) \) can be a degenerate function, the number of variables determining its output can in fact be smaller. In this case we get a lower probability of \( g(x_1, \ldots, x_n, x_p) \) being fully sensitive to \( x_p \).

For some gates, any set of tests detecting all single stuck-at faults on inputs of the gate, also detects all single stuck-at faults on its outputs (e.g. MIN, MAX and literal are such gates). If a circuit is built from such gates only, in order to find a set of tests detecting all single stuck-at faults in the circuit, it is enough to find tests for all primary inputs and all branches of fanout points [6]. Such circuit implementations are particularly suitable for test generation using full sensitivity, since for any internal line \( p \), which is a branch of a fanout point, \( g(x_1, \ldots, x_n, x_p) \) is never a degenerate
function (provided $f(x_1, \ldots, x_n)$ is not a degenerate function). So, for such circuit implementations, it is highly probable that at most $k$ faults are not covered by tests, generated by full sensitivity, where $k$ is the number of fanout branches.

Further research needs to be undertaken to find a more efficient algorithm for computing full sensitivity, based on a more compact representation of functions, for example on multiple-valued decision diagrams [40].
Chapter 6

Composition Trees in Logic Synthesis

In the previous chapter, we consider multiple-valued functions that are fully sensitive to their variables. Such functions were also independently studied by Bernhard von Stengel in [57]. He proved that all such functions have a unique representation, called a composition tree, which suggests the circuit realization of the function at a close to minimal cost. This chapter presents an effective algorithm for generating the composition tree for any function fully sensitive to its variables.

The chapter is organized as follows. Section 6.1 describes the disjunctive decomposition of multiple-valued functions. Section 6.2 summarizes the main results on composition trees. Section 6.3 presents the new algorithm. In section 6.4, we give some conclusions and discuss possible directions for further research.

Some of the results in this chapter are contained in [23].

6.1 Disjunctive decomposition of functions

Generally, the problem of decomposing functions can be formulated as follows. Given a function \( f \), express it as a composite function of some set of new functions. Sometimes, a composite expression can be found in which the new functions are significantly simpler than \( f \). Then the synthesis of a logic circuit realizing \( f \) may be accomplished by synthesizing circuits realizing the simpler functions of the composite representa-
CHAPTER 6. COMPOSITION TREES IN LOGIC SYNTHESIS

...tion, thus reducing the overall cost of implementing $f$.

However, the problem of selecting the "best" decomposition minimizing the cost of a realization of a given function appears to be far too difficult to be solved exhaustively. Therefore, all previous efforts to apply decomposition theory to the synthesis of Boolean and multiple-valued logic circuits restrict the decomposition to be obtained to a particular type. In this chapter we consider disjunctive decompositions only. The basis for the different types of disjunctive decomposition is the simple disjunctive decomposition which is a representation of the form

$$f(x_1, \ldots, x_n) = g(h(x_1, \ldots, x_p), x_{p+1}, \ldots, x_n) \quad (6.1)$$

for some $1 \leq p \leq n$. If $f$, $g$ and $h$ are $m$-valued functions, then in (6.1), the original function $f$ specifying an $n$-input, 1-output $m$-valued circuit is replaced by the specifications of two $m$-valued circuits, one having $p$ inputs and one output, and the other having $1 + n - p$ inputs and one output (see Figure 6.1). If $\Omega_{n,m}$ is an upper bound on the cost of realizing an $m$-valued function of $n$ variables, then the total cost of realizing these two circuits is bounded above by $\Omega_{p,m} + \Omega_{(1+n-p),m}$. Because the cost bound $\Omega_{n,m}$ usually increases nearly exponentially with $n$ [55], the discovery of any nontrivial decomposition of the form (6.2) greatly reduces the cost of realizing $f$.

![Figure 6.1: Simple disjunctive decomposition](image)

The argument of the function $h$ in (6.1) may be any tuple $x = (x_i)_{i \in A}$ of variables for a nonempty subset $A \subseteq N$. We denote the remaining variables by $y = (x_i)_{i \in \overline{A}}$, where $\overline{A} = A \cap N$. So, the variables of $f$ can be partitioned and rearranged as $(x, y)$. 
CHAPTER 6. COMPOSITION TREES IN LOGIC SYNTHESIS

Then a more general form of a simple disjunctive decomposition can be written as

\[ f(x, y) = g(h(x), y) \]  \hspace{1cm} (6.2)

for all values of \( x \) from \( M^A \) and all values of \( y \) from \( M^{\bar{A}} \) and suitable functions \( h: M^A \to M \) and \( g: M \times M^{\bar{A}} \to M \). Any set of variables \( x \) such that \( f \) has a decomposition (6.2) is called a bound set for \( f \). Such a decomposition exists trivially for \( x \) given by any singleton set \( \{x_i\} \) or the all-set \( \{x_1, x_2, \ldots, x_n\} \).

The notion of a bound set is fundamental in decomposition theory. The classical method for recognizing a bound set is based on representing the function by a decomposition chart [1], [13]. The decomposition chart for \( f(x, y) \) is a two-dimensional table where the columns represent the variables from the set \( x \) and the rows the variables from the set \( y \). Then \( x \) is a bound set if and only if the chart has column multiplicity at most \( m \), i.e. there are at most \( m \) distinct columns in the chart [32]. Figure 6.2 shows such a chart for the \( \{x_3\}|\{x_1, x_2\} \) the partitioning of variables of an example 3-valued 3-variable function, where the set \( \{x_1, x_2\} \) is indeed a bound set.

\[
\begin{array}{c|cccc|cccc|ccc}
 x_1 & x_2 & 00 & 01 & 02 & 10 & 11 & 12 & 20 & 21 & 22 \\
\hline
 x_3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
   & 1 & 0 & 1 & 2 & 2 & 1 & 1 & 0 & 1 & 0 \\
   & 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Figure 6.2: Decomposition chart for an example function in \( \Sigma \)

Once a decomposition of type (6.2) has been selected, either \( g, h \), or both may be similarly decomposed, giving one of the following complex disjunctive decomposition types [32]:

\[
\begin{align*}
\text{multiple:} \quad f(x, y, z) & = g(h(x), k(y), z) \\
\text{iterative:} \quad f(x, y, z) & = g(h(k(x), y), z) \\
\end{align*}
\hspace{1cm} (6.3)
\]

or more generally tree-like decompositions as in \( f(x, y, z, w) = g(h(k(x), y), l(z), w) \).

Clearly, since each decomposition of type (6.2) reduces the cost of implementing \( f \), the more \( f \) is decomposed, the more the cost is reduced. However, sometimes
a function can be decomposed in several different ways, depending on the bound set chosen, e.g. when two decompositions of the same function \( f(x, y) = g(h(x), y) \) and \( f(z, v) = k(l(z), v) \) exist such that \( x \cap z \neq \emptyset \). We call such decompositions *conflicting*. Therefore, at this point, a theory is needed to decide which bound sets should be chosen to obtain the "most decomposed" representation of \( f \). Such a theory was developed by Ashenhurst [1] for the case of Boolean functions. He proved that any \( n \)-variable Boolean function that is nondegenerate, i.e. which actually depends on all \( n \) variables to determine its output, has a *composition tree*, which is a representation reflecting any bound set of variables, thus a "most decomposed" one. Hence, the realization of the given function in correspondence with its composition tree (with suitable assumption about the cost of logic elements) should have a cost that is close to minimal. In the sixties it was even conjectured that such an implementation must be a minimal one. However, Paul [46] found a counterexample demonstrating a circuit derived by other than decomposition techniques that has smaller cost than the one implementing the composition tree. Such examples seem to be very rare.

The work of Ashenhurst was generalized by von Stengel to a certain class \( \Sigma \) of general \( n \)-ary operations on (not necessarily finite) sets [57]. Here we consider the theory developed in [57] for a restricted case of homogeneous multiple-valued functions of type \( f : M^n \rightarrow M \). For this case, the class \( \Sigma \) can be defined as follows.

**Definition 6.1** [57] \( \Sigma \) is the class of functions \( f : M^n \rightarrow M \) for some \( n \geq 1 \) that are fully sensitive to all their variables.

Von Stengel has proved that \( \Sigma \) is closed under composition and decomposition, i.e. if \( g \) and \( h \) in \( f(x, y) = g(h(x), y) \) belong to \( \Sigma \), then so does \( f \), and vice versa. The class \( \Sigma \) does not include all functions for which the composition tree exists (compare that, by analogy, full sensitivity doesn’t allow detection of all detectable faults). However, as proved in Chapter 5, the percentage of \( n \)-variable \( m \)-valued functions which are not in \( \Sigma \) is extremely small for large \( n \) and \( m \).
6.2 Composition trees

This section gives the main results on composition trees from [57], necessary for understanding the algorithm developed in the next section.

For convenience, we let a bound set be represented by the indices of the variables, not the variables themselves. So, if \( f(x, y) = g(h(x), y) \) and the indices of variables included in the bound set \( x \) are in \( A \subseteq \mathcal{V} \), we say that \( A \) is a bound set for \( f \).

A function \( f \) that has only the trivial bound sets \( \mathcal{V} \) and \( \{ i \} \) for \( i \in \mathcal{V} \) is called non-decomposable or prime, as, for example, always when \( n \leq 2 \). If \( A \subseteq \mathcal{V} \) is a nontrivial bound set and (6.2) holds, then other bound sets that are subsets or supersets of \( A \) or disjoint to \( A \) relate to decompositions of \( h \) and \( g \):

**Lemma 6.2** [57] Let \( A \) be a bound set with decomposition \( f(x, y) = g(h(x), y) \), where \( h: M^A \to M \) and \( g: M^{(i) \cup \overline{A}} \to M \) for some \( i \in A \). Then for all sets \( C \subseteq A, D \subseteq \overline{A} \):

(a) \( C \) is bound for \( f \) \iff \( C \) is bound for \( h \).

(b) \( \overline{A} \cup D \) is bound for \( f \) \iff \( \{ i \} \cup D \) is bound for \( g \).

(c) \( D \) is bound for \( f \) \iff \( D \) is bound for \( g \).

Several bound sets for \( f \), as in Lemma 6.2, lead to iterative or multiple decompositions of \( f \) as in (6.3). Call a bound set \( A \) for \( f \) strong if any other bound set is either a subset or superset of \( A \) or disjoint to \( A \). For example, the trivial bound sets \( \mathcal{V} \) and \( \{ i \} \) are strong. The partial order of inclusion among these strong bound sets defines a tree. This tree with strong bound sets as nodes (suitably labeled) is called the composition tree of \( f \).

Each node of the composition tree is labeled with a function that has as many variables as the node has children. Leaves are labeled with unary functions, which may be the identity. The hierarchical term of these functions represents \( f \). For a tree such as that shown in Figure 6.3 (which is described more fully below), this term may be

\[
f(x_1, \ldots, x_6) = g(h(a(x_1, x_2), x_3, x_4), x_5, x_6).
\] (6.4)
If all these functions are prime, then all bound sets are strong and the composition tree is fully described. The interesting case is therefore when some bound sets are overlapping with others.

**Theorem 6.3** [57] Let \( f \in \Sigma \) and \( A, B \) be bound sets for \( f \) that overlap, i.e. \( C = A - B \), \( D = A \cap B \), and \( E = B - A \) are not empty. Then

(a) \( A \cup B \) and \( C, D, E \) are bound sets.

(b) \( f \) has a representation \( f(x, y, z, w) = g(a(x) \cdot b(y) \cdot c(z), w) \) for \( x \in M^C \), \( y \in M^D \), \( z \in M^E \) and remaining variables \( w \) where \( \cdot \) is an associative function in \( \Sigma \). which is commutative if and only if \( C \cup E \) is a bound set.

In Theorem 6.3, \( A \) and \( B \) are not strong bound sets, but \( A \cup B \) and its partition classes \( C, D, E \) may be. In that case, the node \( A \cup B \) of the composition tree is labeled with the function \( a \cdot b \cdot c \) of the three variables \( a, b, c \) (with values in \( M \)). In fact, it suffices to store with the node \( A \cup B \) the binary function \( \cdot \) (with \( m^2 \) values) and to specify a **linear order** among its children showing how to take the product with \( \cdot \), like \( a \cdot b \cdot c \) since the order \( b \cdot a \cdot c \) is not allowed if \( \cdot \) is not commutative. If \( \cdot \) is commutative, then the order of taking the product is irrelevant. For \( m > 2 \), \( \Sigma \) may have non-commutative associative operations \( \cdot \) like

\[
\cdot = \begin{cases} a & \text{if } b = 0 \\ b & \text{if } b > 0. \end{cases}
\]
For example, $f$ in (6.4) may also have the overlapping bound sets $\{1,2,3\}$ and $\{3,4\}$, which are bound sets for $h$ by Lemma 6.2(a). Then Theorem 6.3 asserts
\[
h(a(x_1, x_2), x_3, x_4) = \hat{h}(a(x_1, x_2), b(x_3), c(x_4))
\]
(6.5)
which in general are necessary for this representation (but would be unnecessary if $h$ was prime). The functions $h$ and $\hat{h}$ in (6.5) are called isotopic since they are identical except for such bijections $M \to M$ of variable or function values. Isotopy leaves decompositions invariant: if $f$, $g$, or $h$ in (6.2) is replaced by an isotopic function, then the other functions can be replaced by isotopic functions such that (6.2) still holds. As a stronger notion, two. say. binary operations $\ast$ and $\bullet$ are called isomorphic if there is a bijection $\phi: M \to M$ such that $\phi(a \ast b) = \phi(a) \bullet \phi(b)$.

A maximal bound set is an inclusion-maximal bound set not equal to $\emptyset$. Then either
\[
\text{all maximal bound sets are pairwise disjoint.} \quad (6.6)
\]
or
\[
\text{two maximal bound sets } A, B \text{ overlap.} \quad (6.7)
\]
In (6.7), $A \cup B = \emptyset$ because of Theorem 6.3(a). Starting with this case distinction, one can show the following.

**Theorem 6.4** [57] Let $T(f)$ be the composition tree of $f$ given by the strong bound sets as nodes, related by inclusion. Any node of the tree can be labeled "disjoint" (which holds for all nodes with at most two children) or "full" or "linear", such that $A$ is a bound set for $f$ if and only if
(a) $A$ is a node of the tree, or
(b) $A$ is the union $\bigcup_{i \in L} B_i$ of the children $B_1, \ldots, B_k$ of a "full" node, $\emptyset \neq L \subseteq \{1, \ldots, k\}$, or
(c) $A$ is the union $\bigcup_{i=j}^l B_i$ of an interval of the children $B_1, \ldots, B_k$ (specified in linear order) of a "linear" node, $1 \leq j \leq l \leq k$. 

The children of a "disjoint" node \( A \) are the maximal, pairwise disjoint bound sets contained in \( A \) as in (6.6). We assume \( k \geq 3 \) in (b) and (c) to distinguish these cases from a "disjoint" node. The number of bound sets may be exponential, as in (b), but they are efficiently coded by this labeled tree, which has size linear in \( n \) [42].

**Theorem 6.5 [57]** Let \( T(f) \) be the composition tree of \( f: M^N \to M \) and \( B_1, \ldots, B_k \) be the children of the root \( N \). Then

\[
f(y_1, \ldots, y_k) = g(h_1(y_1), \ldots, h_k(y_k))
\]

(6.8)

for functions \( h_i : M^{B_i} \to M \) (1 \( \leq i \leq k \)) and \( g: M^k \to M \) in \( \Sigma \) where

(a) \( g \) is prime if \( N \) is labeled "disjoint".

(b) \( g(a_1, \ldots, a_k) = a_1 \bullet \cdots \bullet a_k \) (for \( a_i \in M \). 1 \( \leq i \leq k \)) with an associative and commutative operation in \( \Sigma \) if \( N \) is labeled "full".

(c) \( g(a_1, \ldots, a_k) = a_1 \bullet \cdots \bullet a_k \) with an associative and non-commutative operation in \( \Sigma \) if \( N \) is labeled "linear".

(d) In (a), \( g \) is unique up to isotopy. In (b) and (c), \( \bullet \) is unique up to isomorphy.

This is the main representation theorem of [57]. It is applied by induction, which ends if \( f \) is prime. Using the trees \( T(h_1), \ldots, T(h_k) \) with the children \( B_1, \ldots, B_k \) of \( N \) as roots, (6.8) gives the fully decomposed representation of \( f \). By Theorem 6.4, the resulting hierarchical term contains any decomposition (6.2) as a subterm, where \( h(x) \) is possibly obtained by suitable arrangement of "products" with an operation \( \bullet \).

An example of a composition tree is shown in Figure 6.3. Abbreviations "D", "F" and "L" stand for labels "disjoint", "full" and "linear", respectively. The linear order among the children of the "L" node is from left to right. Letters \( a, b, c, d, e \) denote the functions associated with the nodes, and \( \bullet \) and \( \circ \) denote the operations. In accordance with the tree, the complete decomposition of the function \( f \) is

\[
f(x_1, \ldots, x_6) = (a(x_1, x_2) \circ b(x_3) \circ c(x_4)) \bullet d(x_5) \bullet e(x_6)
\]
with \( \bullet \) being an associative and commutative operation and \( \circ \) being associative and non-commutative.

The results of [57] prove the existence of the composition tree for a particular class of functions and give a description of this tree. However, this is not algorithmic and does not describe how to build the composition tree from a specification of the given function. In the next section we present a recursive algorithm for generating composition trees of functions in \( \Sigma \). It is a combined top-down and depth-first construction of the tree. In parts, it generalizes Curtis' approach [13] for Boolean functions to the \( m \)-valued case. A bottom-up construction that could be adapted to our situation is described in [42].

6.3 An algorithm for constructing composition trees

In this section we show how to construct the composition tree \( T(f) \) for an \( m \)-valued \( n \)-variable function \( f \in \Sigma \).

The problem of finding a simple disjunctive decomposition, i.e. determining a bound set for an \( m \)-valued function \( f \) and obtaining suitable functions \( g \) and \( h \) in \( f(x, y) = g(h(x), y) \). is widely studied with a number of algorithms developed for its solution including those in [39], [41], [64]. So we assume the existence of the following function:

\[
\text{IsBoundSet}(A, f, N)
\]

input: \( f \in \Sigma, A \subseteq N \)

output: "true" if \( A \) is a bound set for \( f \),

"false" otherwise.

We assume this function has worst-case time complexity \( m^n \). One may hope that the output "false" is produced faster, if more than \( m \) columns in the decomposition chart are detected early. As long as all arguments of \( f \) have to be evaluated, \( m^n \) is the worst-case time complexity [42]. It might be interesting to analyze the expected
time needed to recognize that a "random" function is prime, with all caveats that such functions are not those used in practice.

We check successively all sets \( A \) with cardinality \( n - 1, n - 2, \ldots, 2 \) until a bound set is found. If \( f \) is prime, this requires \( 2^n - n - 2 \) many calls to IsBoundSet and overall time complexity \( O(2^n m^n) \). However, this is the worst-case time complexity. It will become apparent that there is a significant speedup as soon as decompositions are found.

If \( f \) is prime, then we return

\[
\text{TrivialTree}(f, N)
\]

which is just one node if \( N \) is a singleton, otherwise root \( N \) with children \( \{i\} \) for \( i \in N \). The root is labeled "disjoint" and with the function \( f \) to take the place of \( g \) in (6.8), and the children \( B_i = \{i\} \) are labeled with \( h(x_i) = x_i \).

If a bound set \( A \) is found, then it is maximal since we examine the largest subsets of \( N \) first. Then, we construct a specification of \( h \) and \( g \) in (6.2).

If \( A \) is not overlapping with any other bound set, then \( A \) is by definition a node of the composition tree \( T = T(f) \). Using Lemma 6.2, we can recursively compute

\[
T_1 = T(h), \quad T_2 = T(g)
\]

and return

\[
T = \text{Append}(T_1, A, T_2, \{i\}, \overline{A}).
\]

The function Append takes two trees, \( T_1 \) with root \( A \) and \( T_2 \) with root \( \{i\} \cup \overline{A} \), as input and returns \( T \) obtained by replacing the leaf \( \{i\} \) of \( T_2 \) by \( T_1 \) and new root \( A \cup \overline{A} \). The labels of the nodes are not changed. Then Lemma 6.2 and Theorem 6.5 imply \( T = T(f) \).

However, this is not correct if another bound set \( B \) overlaps with \( A \), as in (6.7). Then \( A \cup B = N \) because \( A \) is maximal, hence \( \overline{A} = N - A = B - A \) and Theorem 6.3(a) implies:
A - B and A ∩ B are bound sets for f.

\( \overline{A} \) is a bound set for f.

By Lemma 6.2(a) and (c), these necessary conditions are equivalent to

A - B and A ∩ B are bound sets for h.

\( \overline{A} \) is a bound set for g.

As before, these conditions can be verified from the composition trees \( T_1 \) and \( T_2 \) of h and g in (6.9) if these are computed recursively first. The conditions fail if the root of either tree is labeled "disjoint" and has three or more children, by (6.6). Otherwise, we invoke

**PossiblyMerge**\( (T_1, A, T_2, \{i\}, \overline{A}) \)

input: composition trees \( T_1 = T(h) \) with root A and \( T_2 = T(g) \) with root \( \{i\} \cup \overline{A} \).

Assume (6.2).

output: If no bound set \( B \) for f overlaps with A, then the same as **Append**\( (T_1, A, T_2, \{i\}, \overline{A}) \).

Otherwise, \( T = T(f) \) with the roots of \( T_1 \) and \( T_2 \) merged, the leaf \( \{i\} \) of \( T_2 \) omitted, and new root \( A \cup \overline{A} \) labeled "full" or "linear".

We describe this procedure in more detail. The root of \( T_2 = T(g) \) is labeled "disjoint" since \( \{i\} \) is a maximal bound set for g (by Lemma 6.2(b), since A is maximal for f). Suppose the root of \( T_2 \) has two children \( \{i\} \) and \( \overline{A} \), and the root of \( T_1 = T(h) \) has two children \( C, D \). Then the functions \( G \) and \( H \) for these roots (which are stored with the trees) show

\[
\begin{align*}
g(h, y) &= G(h, c(y)), \quad h \in M, \ y \in M^\overline{A} \\
h(u, v) &= H(a(u), b(v)), \quad u \in M^C, \ v \in M^D
\end{align*}
\]

so that \( f(u, v, y) = G(H(a(u), b(v)), c(y)) \). We have to check for the possibility that \( G \) and \( H \) are isotopic to an operation \( \bullet \). Rather than trying out the bijections \( M \to M \) for verifying such an isotopy, we test if \( B = D \cup \overline{A} \) (i.e. the variables \( v, y \)) and \( B = C \cup \overline{A} \) (i.e. the variables \( u, y \)) are bound sets for \( f(u, v, y) \). It is not necessary
CHAPTER 6. COMPOSITION TREES IN LOGIC SYNTHESIS

to apply this test to \( f \). Equivalently, we test if \( \{b, c\} \) and \( \{a, c\} \) are bound sets for the function \( F(a, b, c) = G(H(a, b), c) \) of the three variables \( a, b, c \) with values in \( M \). This can be done quickly in time \( m^3 \). Then if

(i) \( \{b, c\} \) and \( \{a, c\} \) are bound: merge the roots and label the new root \( A \cup \overline{A} \) "full". with children \( C, D, \overline{A} \).

(ii) \( \{b, c\} \) is bound, \( \{a, c\} \) is not: merge, with label "linear" and children \( C, D, \overline{A} \).

(iii) \( \{a, c\} \) is bound, \( \{b, c\} \) is not: merge, with label "linear" and children \( D, C, \overline{A} \).

(iv) Otherwise, just append \( T_1 \) to \( T_2 \).

In cases (i)--(iii), the root obtained by merging is labeled with the operation \( \bullet \). Furthermore, it may be necessary to apply a bijective transformation to the values of the functions of the children, as when changing from \( h \) to \( \hat{h} \) in (6.5).

PossiblyMerge is also applied if the root of \( T_1 \) is labeled "full" or "linear" with children \( B_1, \ldots, B_k \). In that case, we let \( C = B_1 \) and \( D = B_2 \cup \cdots \cup B_k \), and let \( H \) be the operation \( \bullet \) at the root of \( T_1 \), so that (6.10) holds and proceed as before. In case (ii), the children of the new root are \( B_1, \ldots, B_k, \overline{A} \). In case (iii), they are \( B_k, \ldots, B_1, \overline{A} \). In any case, PossiblyMerge has time complexity \( O(m^3) \).

The great advantage of the recursion (6.9) is that it saves computation time. If \(|A| = p\) as in Figure 6.3, each test of a bound set for \( h \) or \( g \) with IsBoundSet requires up to \( m^p \) or \( m^{1+n-p} \) many steps, much fewer than the \( m^n \) steps for \( f \).

For the computation of \( T_2 = T(g) \), some care is necessary to avoid duplicate computations. First, \( \{i\} \) is a leaf of \( T_2 \), so only subsets of \( \overline{A} \) have to be checked. Second, subsets \( D \) of \( \overline{A} \) with \( |D| > p \) can also be disregarded since they have already been checked for \( f \) and Lemma 6.2(c) holds. The second point is relevant only when \( p \leq |\overline{A}| = n - p \), i.e. \( p \leq n/2 \). In that case (which includes \( p = |\overline{A}| \)) it is also unnecessary to invoke PossiblyMerge since then \( |B| > p \) for the bound set \( B \) of \( f \) that is sought there, which is not possible.

In order to compute \( T_2 \) efficiently, we therefore pass \( S := \overline{A} \) and \( p \) as additional parameters to the algorithm, which looks as follows. Preconditions and assertions
at various stages are given in angle brackets (...). The procedure terminates at each return statement with the indicated output.

CompositionTree(f, N, S, p)
input: f ∈ Σ, S ⊆ N, integer p
assumption: (A ⊆ S and |A| ≤ p for any bound set A of f, A ≠ N)
output: the composition tree T = T(f)
initial call: CompositionTree(f, N, N, n − 1)

1. if |N| ≤ 2 then
   return T := TrivialTree(f, N):
2. while p > 1
   for all A ⊆ S with |A| = p
      if IsBoundSet(A, f, N) then goto 3:
   end for:
   p := p − 1:
end while:
⟨ f is prime ⟩
return T := TrivialTree(f, N):
3. ⟨ A is a bound set for f, |A| = p ⟩
   let f(x, y) = g(h(x), y) as in (6.2), i ∈ A
   T1 := CompositionTree(h, A, A, p − 1):
   T2 := CompositionTree(g, {i} ∪ (N − A), S − A, min{p, |S − A|});
4. if N ≠ S or p ≤ |N| − p or T1 or T2 has a “disjoint” root with > 2 children
   then
   T := Append(T1, A, T2, {i}, N − A)
   else
   T := PossiblyMerge(T1, A, T2, {i}, N − A);
return T.
CHAPTER 6. COMPOSITION TREES IN LOGIC SYNTHESIS

When computing $T_2$, we exploit that $T_2$ has a "disjoint" root when we invoke CompositionTree with third parameter $S := S - A$ rather than $N - A$, since all elements of $N - A$ (as singletons) will be children of the root. For example, suppose $N = 12345678$ (as shorthand for $\{1, \ldots, 8\}$ with disjoint maximal bound sets 123, 45, 67, 8. Assume $i \in A$ in step 3 is the first element of $A$. Then the parameters $N, S$ of CompositionTree for computing $T_2$ are

- after 123 is found: $N = 145678, S = 45678$.
- after 45 is found: $N = 14678, S = 678$.
- after 67 is found: $N = 1468, S = 8$.

Similarly, the test $N \neq S$ in step 4 reveals if the current computation is for some tree $T_2$.

To illustrate the recursive calls for computing $T_1$, consider Figure 6.3 where $N = 123456$. In succession, we find the bound sets 12345, 1234, 123, 12. (In this example, $T_2$ is therefore always the trivial tree by step 1.) Let $k(x_1, x_2, x_3)$ be the function with root 123, $k(x_1, x_2, x_3) = K(a(x_1, x_2), x_3)$. Because $K$ and $a$ are binary functions. PossiblyMerge($T(a), \{1, 2\}, T(k), \{1\}, \{3\}$) is called, which is the same as Append since (case (iv) above) neither $\{2, 3\}$ nor $\{1, 3\}$ are bound sets for $k$. After this recursion terminates, the function $l$ for node 1234 is checked with, say. PossiblyMerge($T(k), \{1, 2, 3\}, T(L), \{1\}, \{4\}$) which merges 123 into 1234 (case (ii) above), making this a "linear" node. Next, via PossiblyMerge, case (iv), 1234 is just appended to 12345, and then 12345 is merged via case (i) into 123456 which is labeled "full".

The algorithm exploits the structure of bound sets as stated in Theorem 6.3 and Theorem 6.4, and there seems to be no obvious way to do this better. Furthermore, it is readily adapted to bound sets $A$ that are apparent from a modular specification of the function, where $h$ and $g$ in (6.2) are explicitly given. After computing $T_1 = T(h)$ and $T_2 = T(g)$, these trees could possibly be merged. The above procedure PossiblyMerge can easily be modified for that purpose so that it works with arbitrary
CHAPTER 6. COMPOSITION TREES IN LOGIC SYNTHESIS

composition trees $T_2$.

Making use of decompositions that are found along the way reduces the running time substantially. If $f$ is composed only of binary functions, for example, then $f$ has a bound set of size $n - 1$, which is found after $nm^n$ or fewer steps, and the same holds for any function $h$ (with correspondingly smaller number $n$ of variables). Thus, the complete tree is computed in $\sum_{i=3}^{n} im^i = O(nm^n)$ time. The expensive cost $m^n$ of finding a bound set is still there, but with a factor of $n$ rather than $2^n$ for a non-decomposable function.

6.4 Conclusion

This chapter presents an effective recursive algorithm for generating the composition tree for any function fully sensitive to its variables. If the composition tree of a function is not trivial, then the cost of realizing the function can be reduced by implementing it in correspondence with its composition tree.

If all functions – or at least a large class of $m$-valued functions – were disjunctively decomposable, the algorithm presented in this chapter would have been more than adequate for obtaining highly economical multi-level $m$-valued circuits. However, the fraction of all Boolean functions of $n$ variables possessing nontrivial disjunctive decompositions of type (6.2) approaches zero as $n$ approaches infinity [55, p. 90]. It is straightforward to generalize this result to $m$-valued functions for $m > 2$. This implies that most of the functions in $\Sigma$ have trivial composition trees. However, this does not mean that the disjunctive decomposition theory developed in [1] and [57] is of no practical value. First, the "practical" functions are not randomly distributed through the space of all functions. Second, in the Boolean case, Ashenhurst's disjunctive decomposition theory led to the formulation in [13] of the general theory of nondisjunctive decompositions, a theory encompassing all switching functions on $n$ variables regardless of the size of $n$. We hope that the theory developed in [57] can serve as a base for more general investigations of $m$-valued functions, and that
our algorithm can be used as a starting point for developing a systematic synthesis algorithm capable to find an efficient implementation for any function. The following extensions of the algorithm would be the first steps towards this.

First, the present algorithm can construct the composition tree only for functions which are in $\Sigma$, which does not include all functions with disjunctive decompositions. An open problem remains how to generalize class $\Sigma$ so that it includes all disjunctively decomposable functions.

Second, the functions included in class $\Sigma$ are restricted to homogeneous functions. If the theory developed in [57] can be extended to the case of heterogeneous functions, then it would have a direct application to Boolean circuit synthesis, since it would cover as a special case decompositions of the type

$$f(x, y) = g(h(x), y)$$

with $f: \{0, 1\}^n \to \{0, 1\}$, $h: \{0, 1\}^p \to \{0, 1, \ldots, m - 1\}$ and $g: \{0, 1, \ldots, m - 1\} \times \{0, 1\}^{n-p} \to \{0, 1\}$. In such a decomposition the $m$-valued function $h$ of 2-valued variables can be coded by $k = \lfloor \log_2 m \rfloor$ Boolean functions $h_1, h_2, \ldots, h_k$, giving a decomposition of the form

$$f(x, y) = g(h_1(x), h_2(x), \ldots, h_k(x), y)$$

(6.11)

with all functions being Boolean. The decomposition of type (6.11) includes as a subclass simple disjunctive decompositions ($k = 1$) as well as nondisjunctive decompositions. As long as $f$ is a function of more than three variables, such a decomposition can always be found with $h_1(x), h_2(x), \ldots, h_k(x)$ and $g$ each having fewer arguments than $f$, for there always exists a decomposition of the form

$$f(x_1, \ldots, x_n) = f(z, x_n) = g(h_1(z), h_2(z), x_n)$$

with $z = (x_1, \ldots, x_{n-1})$. Thus, a decomposition (6.11) allows the simplification of any Boolean function. Therefore, a theory of composition trees for this extended case would be a basis for the systematic logic synthesis of multi-level Boolean circuits.
Chapter 7

Synthesis of Easily Testable Circuits

For some applications, finding a minimal circuit realization for a given function might not be the primary goal of logic synthesis. In this chapter, we develop an approach to logic synthesis suitable for the applications in which the ability to test circuits easily and quickly is critical. We consider logic circuits realizing $m$-valued functions in a modulo $m$ sum-of-products canonical form. While this canonical form is extensively studied, its applications to logic synthesis have only been considered for the case $m = 2$. The circuits realizing modulo 2 sum-of-products forms are proved to be easily testable [6]. In this section we investigate the case $m > 2$. Generalizing from the two to the $m$-valued case, however, is shown to be a non-trivial problem because for $m > 2$ several new phenomena occur which allow us to reduce the upper bound on the number of tests required for fault detection but make the generation of tests harder.

The chapter is organized as follows. Section 7.1 introduces the circuits realizing multiple-valued functions in modulo $m$ sum-of-products canonical form. Section 7.2 estimates the number of tests which are needed to detect all single stuck-at faults on the internal lines of a circuit realizing a modulo $m$ sum-of-products form. In Section 7.3, the testability of the primary inputs is investigated. A procedure for test generation for primary inputs is described and its effectiveness is evaluated. In
Section 7.4. we show that by adding an extra multiplication mod \( m \) gate with an observable output to the circuit the number of tests sufficient to detect all single stuck-at faults is reducible to four. Section 3.8 concludes the chapter.

Some of the results in this chapter are contained in [21].

7.1 Implementation of modulo \( m \) sum-of-products canonical form.

A logic function can be implemented with many different circuit designs. Various realizations of the same function may require different numbers of input vectors as tests. For example, a two-level AND-OR realization of the \( n \)-variable two-valued parity function (which has the value 1 if and only if an odd number of the variables have the value 1) requires all \( 2^n \) possible input vectors as tests to detect all single stuck-at faults. However, this function can also be implemented as a multi-level tree of two-input XOR gates, and this realization requires only two tests to detect all single stuck-at faults.

In two-valued systems, testing the multi-level tree of XOR gates is easy because in a fanout free linear circuit, any single fault propagates to the output independently of the input vector applied. This property allows the minimization of the number of tests required for fault detection and simplifies the generation of tests for the whole family of logic circuits, called Reed-Muller circuits. A Reed-Muller circuit realizes a Reed-Muller canonical form of the function (3.1). The circuit consists of a multi-level tree of two-input XOR gates, fed by AND gates (Figure 7.1). Any Reed-Muller circuit can be tested for all single stuck-at faults with a maximum of \( 3n + 4 \) input vectors, where \( n \) is the number of primary inputs [6].

The useful property of a multi-level tree of XOR gates to propagate any single fault to the output remains valid when XOR gates are replaced with sum mod \( m \) gates and more than two levels of signals are used in the circuit. So, an \( m \)-valued circuit based on a multi-level tree of sum mod \( m \) gates might possess the useful property of
easy testability.

In this chapter, we investigate the testability of one such architecture, which is a circuit realizing an \( m \)-valued function in modulo \( m \) sum-of-products (SOP) form (3.3), where \( m \) is a prime. Recall, that modulo \( m \) addition and multiplication form a Galois field of order \( m \) \((GF(m)) \). Throughout this chapter, we assume that \( m \) is a prime greater than two.

A modulo \( m \) SOP form can be implemented by the circuit shown in Figure 7.2. It consists of a linear cascade of two-input sum mod \( m \) gates fed by multiplication mod \( m \) gates, one corresponding to each product-term of the expansion with non-zero constant \( c_i, \quad i \in \{1, \ldots, m^n - 1\} \). The input \( x_0 \) has the value of the constant \( c_0 \) during normal operation and a value different from \( c_0 \) during testing.

For example, the 3-variable 3-valued function

\[
f(x_1, x_2, x_3) = x_1^2 x_2 \oplus 2 x_1 x_2^2 \oplus 1 x_1 x_2 x_3 \oplus 2 x_1 x_2 x_3^2
\]

can be implemented by the circuit shown in Figure 7.3.

As a fault model, we use a single stuck-at fault model. First, we consider detection of internal faults, which occur on the inputs of the individual gates. We prove that
only four tests are sufficient to detect all single stuck-at faults on internal lines in a circuit realizing \( m \)-valued functions in modulo \( m \) SOP form. Furthermore, this set of tests is independent of the function being realized and therefore universal. Second, we give two alternative techniques for the testing of primary inputs - one by generating a test set of maximum length \( 2n \), and the other by adding to the circuit an extra multiplication mod \( m \) gate with an observable output to ensure that the four tests for internal lines also detect all single stuck-at faults on primary inputs.

### 7.2 Testability of internal lines.

It is proved in [48] that in the two-valued Reed-Muller circuit realization of a Boolean function \( f(x_1, \ldots, x_n) \) at most \( n + 4 \) tests are required to detect all internal single stuck-at faults. The proof is constructive by showing that, independently of the function being realized, a set \( T = T_1 \cup T_2 \) detects all internal single stuck-at faults. \( T_1 \) is defined by the table below:

<table>
<thead>
<tr>
<th>( x_0 )</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( \ldots )</th>
<th>( x_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>\ldots</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>\ldots</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>\ldots</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>\ldots</td>
<td>1</td>
</tr>
</tbody>
</table>

It detects all single faults on the inputs of XOR gates and all stuck-at-0 faults on the inputs of AND gates. \( T_2 \) is defined by \( T_2 := \{T_{21}, T_{22}, \ldots, T_{2n}\} \) with the test \( T_{2i} \) having \( x_i = 0 \) and \( x_j = 1 \) for all \( j \neq i, i, j \in N \). It detects all stuck-at-1 faults on the
CHAPTER 7. SYNTHESIS OF EASILY TESTABLE CIRCUITS

inputs of AND gates. So, the $n + 4$ tests in the test set $T = T_1 \cup T_2$ detect all internal single stuck-at faults in a Reed-Muller circuit.

We use a similar approach to prove that, in the modulo $m$ SOP circuit realization of an $m$-valued function, only four tests are required to detect all internal single stuck-at faults. It might appear surprising that the multiple-valued case requires less tests than the two-valued one. Before giving the result, we explain the intuition behind this phenomenon.

Consider an $n$-input multiplication mod $m$ gate $G$ with $m$ being a prime. Let $a_i \in M$ be the value of the input variable $x_i$, for $i \in N$. Since the cancellation law of multiplication holds for $GF(m)$ [2], for any $a, b, c \in M$ we have:

$$ab \neq ac$$

It follows from the above that if an input vector $(a_1, \ldots, a_n)$ such that $a_i \neq 0$ for all $i$, is applied to $G$, then a change in the value of any single input $x_i$ causes a change in the value on the output. In terms of the terminology introduced in Chapter 5, we can say that such an input vector makes the output fully sensitive to $x_i$. This implies that $(a_1, \ldots, a_n)$ is a test for all $x_i$ stuck-at-$\overline{a}_i$ faults, where $\overline{a}_i$ denotes any value but $a_i$, i.e. $\overline{a}_i \in M - \{a_i\}$.

By applying the same reasoning as above, one can see that to detect the remaining stuck-at-$a_i$ faults on each input $x_i$, another input assignment $(b_1, \ldots, b_n)$ such that $b_i \neq 0$ and $a_i \neq b_i$ for all $i$ has to be applied.

So any two input assignments $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ such that none of $a_i, b_i$ is zero and $a_i \neq b_i$ for all $i \in N$, detect all single stuck-at faults on the inputs of a multiplication mod $m$ gate for $m$ being a prime greater than two. It is easy to see why $m = 2$ is an exception. In the two-valued case there is only one input assignment with all entries different from zero, namely $(1 \ldots 1)$.

Since the cancellation law of addition also holds for $GF(m)$, by applying the similar reasoning to that above, we can see that any two input assignments $(a_1, \ldots, a_n)$
and \((b_1, \ldots, b_n)\) such that \(a_i \neq b_i\) for all \(i \in N\). detect all single stuck-at faults on the inputs of an \(n\)-input sum mod \(m\) gate. Notice, that the requirement \(x \neq 0\) is not postulated in the cancellation law of addition, so the entries of the assignments can have the value zero as well. Thus, the above statement holds also for the case \(m = 2\), i.e. for an XOR gate.

We can now give the main result of the section.

**Theorem 7.1** There exists a universal set of four tests which detects all single stuck-at faults on internal lines in a circuit realizing an \(m\)-valued function in modulo \(m\) SOP form.

**Proof:** The proof is constructive. Consider the set \(\mathcal{T}\) consisting of four tests defined by the table below.

<table>
<thead>
<tr>
<th>(x_0)</th>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(\ldots)</th>
<th>(x_n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(\ldots)</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(\ldots)</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(\ldots)</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>(m-1)</td>
<td>(m-1)</td>
<td>(\ldots)</td>
<td>(m-1)</td>
</tr>
</tbody>
</table>

Let us denote by \(i_k\) and \(j_k\) the inputs of the \(k\)th addition mod \(m\) gate in the cascade, as shown on Figure 7.4.

![Figure 7.4: Circuit realizing modulo m SOP form.](image)

1. The first test of \(\mathcal{T}\) results in applying \((0, 0)\) to each pair \((i_k, j_k)\), detecting all stuck-at-0 faults on \(i_k\) and \(j_k\). By stuck-at-\(a\) faults we mean all stuck-at-\(b\) faults for any \(b \in M - \{a\}\).
2. The second test of $T$ results in applying $(\ast, c_k)$ to each pair $(i_k, j_k)$, where $c_k$ is the constant (non-zero) which is fed into the $k$th multiplication mod $m$ gate and $\ast$ denotes any value from $M$. It detects all $j_k$ stuck-at-0 faults. This test also detects all stuck-at-$\overline{1}$ faults on the inputs of the multiplication mod $m$ gates.

3. The third test of $T$ results in applying $(1.0)$ to each pair $(i_k, j_k)$, detecting all $i_k$ stuck-at-0 faults.

4. The fourth test of $T$ applies the value $(m - 1)$ to the inputs of multiplication mod $m$ gates, detecting all stuck-at-1 faults on them.

Hence the four tests completely test the internal lines for all single stuck-at faults.

The above theorem gives us the number of tests which are sufficient to detect all internal single stuck-at faults in a circuit realizing a modulo $m$ SOP form. Since the proof is constructive, it shows how to generate the test set itself. This test set is independent of the function being realized, and therefore universal. In the next section we investigate the testability of the primary inputs.

7.3 Testability of primary inputs.

We show that the number of tests which are sufficient to detect all stuck-at faults on primary inputs in a circuit realizing a modulo $m$ SOP form, as well as in an arbitrary $m$-valued combinational logic circuit realizing a function $f(x_1, \ldots, x_n)$, is at most $2n$. For any variable $x_i$, provided $f(x_1, \ldots, x_n)$ is not degenerate in $x_i$, there exist values $a_1, \ldots, a_n$ and $b_i \neq a_i$ such that

$$f(a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n) \neq f(a_1, \ldots, a_{i-1}, b_i, a_{i+1}, \ldots, a_n) \quad (7.1)$$
Since the change in the value of \( x_i \) from \( a_i \) to \( b_i \) causes a change in the value of \( f \), the input vector \( \tau_1 = (a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n) \) is a test for all \( x_i \) stuck-at faults which set the output of the circuit to a logic value different from \( f(a_1, \ldots, a_n) \). On the other hand, the input vector \( \tau_2 = (a_1, \ldots, a_{i-1}, b_i, a_{i+1}, \ldots, a_n) \) is a test for all \( x_i \) stuck-at faults which set the output of the circuit to \( f(a_1, \ldots, a_n) \). Thus \( T = \{ \tau_1, \tau_2 \} \) is a test set for all single stuck-at faults on \( x_i \). Considering all inputs, a set of \( 2n \) tests for all single stuck-at faults on primary inputs can be obtained.

While it is straightforward to show the existence of a test set of length \( 2n \), the problem of generating this test set is not trivial. The next section presents a procedure for finding \( 2n \) tests for detecting all single stuck-at faults on primary inputs of a circuit realizing a modulo \( m \) SOP form.

### 7.3.1 A procedure for test generation.

It is shown in [48] that in the two-valued case, the number of tests required to detect all single stuck-at faults on primary inputs in a Reed-Muller circuit is \( 2n_e \), where \( n_e \) is the number of primary inputs appearing in an even number of product-terms in the Reed-Muller expansion of the \( n \)-variable function being realized. The following procedure is applied to find the test set. For a primary input \( x_i \), all AND gates having \( x_i \) as input are considered. From these, a gate \( G_i \) with the minimal number of other inputs is selected. Further, two tests, \( \tau_{i1} \) and \( \tau_{i2} \) are defined in the following way:

- \( \tau_{i1} \) specifies \( x_i = 0 \), all other inputs of \( G_i \) to 1, and all other primary inputs to 0.
- \( \tau_{i2} \) specifies \( x_i = 1 \), all other inputs of \( G_i \) to 1, and all other primary inputs to 0.

The test \( \tau_{i1} \) detects \( x_i \) stuck-at-1, and the test \( \tau_{i2} \) detects \( x_i \) stuck-at-0. The procedure is repeated for all \( n \) inputs.

Unfortunately, this simple procedure cannot be used directly for the case \( m > 2 \) for the following reason. A modulo \( m \) SOP form of an \( m \)-valued function \( f(x_1, \ldots, x_n) \) can have \( (m - 1) \) different powers of each variable \( x_i \) involved in the product-terms. If \( m = 2 \), only one power of a variable \( x_i \) is employed, and thus, a single gate \( G_i \) with
the minimal number of other inputs can always be selected. By assigning all but \(x_i\) inputs of \(G_i\) to value 1, and all other primary inputs to 0, a single path from \(x_i\) to the output is sensitized. So, the effect of a fault on \(x_i\) is always propagated to the output. In a circuit realizing a modulo \(m\) SOP form, there may be more than one multiplication mod \(m\) gate depending on \(x_i\) and \(k\) other primary inputs. If these \(k\) primary inputs are assigned to 1 and the rest of the primary inputs to 0, then the effect of a fault on \(x_i\) is propagated along multiple paths, and thus may be canceled out by the sum mod \(m\) cascade. Therefore, in the case of circuits realizing modulo \(m\) SOP forms, all \(m^k\) possible combinations of values for \(k\) primary inputs, not assigned to 0, should be examined to find out which one makes the output sensitive to \(x_i\). Such an assignment always exists, provided the circuit doesn’t have redundant multiplication mod \(m\) gates.

As an illustration, consider the circuit in Figure 7.3 and suppose we generate tests for the primary input \(x_1\). All four multiplication mod \(m\) gates have \(x_1\) as input, but the first and the second gates depend on the minimal number of other primary inputs (\(x_2\) only). If we set \(x_2 = 1\) and \(x_3 = 0\), then the circuit implements the function \(f(x_1, 1, 0) = 1x_1^2 \oplus 2x_1^2 = 0\), i.e. the output is not sensitive to \(x_1\). However, for the input assignment \(x_2 = 2\) and \(x_3 = 0\), the circuit implements the function \(f(x_1, 2, 0) = 2x_1^2 \oplus 2x_1^2 = 1x_1^2\), and thus the output is sensitive to \(x_1\).

Summarizing, the modified procedure for finding the test set of size \(2n\) for detecting all single stuck-at faults on primary inputs of a circuit realizing a modulo \(m\) SOP form is as shown below.

**Procedure for test generation for primary inputs of a circuit realizing a modulo \(m\) SOP form:**

1. Consider all multiplication mod \(m\) gates having \(x_i\) as input:

2. From these, select the gates depending on the minimal number of other primary inputs. Define a set \(X := \{x_j \mid x_j\text{ is a primary input on which all selected gates}\)
3. With \( x_j = 0 \) for all \( x_j \notin X \), find an assignment \( A \) for the primary inputs in \( X - \{ x_i \} \) under which the output is sensitive to the input \( x_i \), i.e. for some values \( a_i \neq b_i \) the transition in the value of \( x_i \) from \( a_i \) to \( b_i \) causes a change in the value of the output. The simplest way to find such an assignment depends on the mechanism used to specify the circuit:

4. Define two tests \( r_{i1} \) and \( r_{i2} \) in the following way:

- \( r_{i1} \) specifies \( x_i = a_i \). primary inputs in \( X - \{ x_i \} \) in correspondence with the assignment \( A \), and \( x_j = 0 \) for all \( x_j \notin X \).
- \( r_{i2} \) specifies \( x_i = b_i \). primary inputs in \( X - \{ x_i \} \) in correspondence with the assignment \( A \), and \( x_j = 0 \) for all \( x_j \notin X \).

5. Repeat the procedure for all primary inputs.

The complexity of the procedure depends on the size of \( X \). The smaller the size of \( X \), the easier it is to find the assignment \( A \). In the simplest case when \( |X| = 1 \), there is a multiplication mod \( m \) gate(s) in the circuit depending on primary input \( x_i \) only (i.e. realizing some power of \( x_i \)). Then, to generate tests for \( x_i \) stuck-at faults, only \( m \) values of \( x_i \) should be examined to find \( a_i \) and \( b_i \) satisfying:

\[
f(0, \ldots, 0, a_i, 0, \ldots, 0) \neq f(0, \ldots, 0, b_i, 0, \ldots, 0)
\]

In the next section we show that for a random circuit implementing an \( m \)-valued \( n \)-variable function in modulo \( m \) SOP form, the probability that \( |X| \leq 2 \) is greater than 99.99% for any \( x_i \), provided \( n \geq 3 \) and \( m \geq 3 \).

### 7.3.2 Evaluation of the effectiveness of the procedure.

As shown in the previous section, it is easy to find a test for a primary input \( x_i \) of a circuit realizing a modulo \( m \) SOP form if the circuit has a multiplication mod \( m \) gate
depending on \( x_i \) and a small number \( k \) of other primary inputs. In this section we estimate how often this is the case. Lemma 7.2 and Lemma 7.3 give the mathematical foundation of the result.

**Lemma 7.2** In the modulo \( m \) SOP form of an \( m \)-valued \( n \)-variable function, the number \( \Theta_{k,n} \) of \( k \)-variable product-terms which include a given variable \( x_i \) is at most:

\[
\Theta_{k,n} = (m - 1)^k \frac{(n - 1)!}{(n - k)!(k - 1)!}
\]

where \( 1 \leq k \leq n \) and \( i \in \mathcal{V} \).

**Proof:** In a modulo \( m \) SOP form, each variable can have \((m - 1)\) different powers. Thus, there can be constructed \((m - 1)^k\) different product-terms consisting of \( k \) fixed variables. Since the number of choices of \((k - 1)\) variable from \((n - 1)\) is \( \binom{n - 1}{k - 1} \), the maximum number \( \Theta_{k,n} \) of \( k \)-variable product-terms which include a given variable \( x_i \) is:

\[
\Theta_{k,n} = (m - 1)^k \binom{n - 1}{k - 1} = (m - 1)^k \frac{(n - 1)!}{(n - k)!(k - 1)!}.
\]

\( \square \)

For example, for \( m = 3, n = 3 \) and a variable \( x_1 \) we have:

\[
\begin{align*}
  k = 1 & : \quad \Theta_{1,3} = 2^1 \begin{pmatrix} 2 \\ 0 \end{pmatrix} = 2, \quad \text{product-terms: } x_1, x_1^2 \\
  k = 2 & : \quad \Theta_{2,3} = 2^2 \begin{pmatrix} 2 \\ 1 \end{pmatrix} = 8, \quad \text{product-terms: } x_1x_2, x_1x_2^2, x_1^2x_2, x_1^2x_2^2, x_1x_3, x_1x_3^2, x_1^2x_3, x_1^2x_3^2 \\
  k = 3 & : \quad \Theta_{3,3} = 2^3 \begin{pmatrix} 2 \\ 2 \end{pmatrix} = 8, \quad \text{product-terms: } x_1x_2x_3, x_1x_2x_3^2, x_1x_2^2x_3, x_1^2x_2x_3, x_1^2x_2x_3^2, x_1^2x_2^2x_3, x_1^2x_2^2x_3^2
\end{align*}
\]

**Lemma 7.3** The fraction \( \Phi_{k,n} \) of modulo \( m \) SOP forms of \( m \)-valued \( n \)-variable functions not having a product-term of \( k \) or less variables which include a given variable \( x_i \) is:
\[ \Phi_{k,n} = \frac{1}{m^{(\sum_{i=1}^{k} \Theta_{i,n})}} \]

where \( 1 \leq k \leq n \) and \( i \in N \).

**Proof:** Since \( m^n \) is the maximum number of different product-terms in a modulo \( m \) SOP form of an \( m \)-valued \( n \)-variable function, and \( \Theta_{k,n} \) is the maximum number of \( k \)-variable product-terms which include a given variable \( x_i \), the number of the modulo \( m \) SOP forms not having a product-term of \( k \) or less variables which include a given variable \( x_i \) is

\[ m^{(m^n - \sum_{i=1}^{k} \Theta_{i,n})} \]

So, the fraction of the modulo \( m \) SOP forms of \( m \)-valued \( n \)-variable functions not having a product-term of the latter type is

\[ \frac{m^{(m^n - \sum_{i=1}^{k} \Theta_{i,n})}}{m^{m^n}} = \frac{1}{m^{(\sum_{i=1}^{k} \Theta_{i,n})}} \]

\[ \square \]

For example, for \( m = 3, n = 3 \) and a variable \( x_i, i \in \{1, 2, 3\} \), we have:

\[ \Phi_{1,3} = 1/3^2 \approx 0.11 \]
\[ \Phi_{2,3} = 1/3^{10} \approx 1.69 \times 10^{-5} \]
\[ \Phi_{3,3} = 1/3^{18} \approx 2.89 \times 10^{-9} \]

The result \( \Phi_{2,3} \approx 1.69 \times 10^{-5} \) implies that for \( m = 3, n = 3 \) and a given primary input \( x_i \), the percentage of circuits realizing modulo \( m \) SOP forms not having a multiplication mod \( m \) gate realizing \( x_i^k \) or \( x_i^k x_j^p \), where \( x_j \) is some other primary input and \( k \) and \( p \) are some powers of \( x_i \) and \( x_j \), is extremely small. Since the fraction \( \frac{1}{m^{(\sum_{i=1}^{k} \Theta_{i,n})}} \) decreases as \( m \) and \( n \) increase, for larger values of \( m \) and \( n \) the value of \( \Phi_{2,n} \) becomes even smaller. So, for a random circuit implementing an \( m \)-valued \( n \)-variable function in modulo \( m \) SOP form, the probability that \( |X| \leq 2 \) is greater than 99.99\% for any \( x_i \), provided \( n \geq 3 \) and \( m \geq 3 \).
In the next section we show that by adding to the circuit an extra multiplication mod \( m \) gate with an observable output the number of tests sufficient to detect all single stuck-at faults is reducible to four.

### 7.4 Testability by hardware redundancy.

It is proved in [48] that, by providing a two-valued Reed-Muller circuit with an extra AND gate having an observable output, \( n + 4 \) tests for internal lines also detect all single stuck-at faults on primary inputs. We show that a similar technique can be used to ensure that the four tests for internal lines given by Theorem 7.1 also detect all single stuck-at faults on primary inputs of a circuit realizing a modulo \( m \) SOP form.

![Circuit with an extra multiplication mod m gate](image)

**Figure 7.5:** Circuit with an extra multiplication mod \( m \) gate \( G^* \).

Consider a circuit realization of an \( m \)-valued function \( f(x_1, \ldots, x_n) \) having an extra multiplication mod \( m \) gate \( G^* \) depending on all input variables \( x_1, \ldots, x_n \) and with an output \( g \) (Figure 7.5). If \( g \) is also observable, then two input assignments \((a_1, \ldots, a_n)\) and \((b_1, \ldots, b_n)\), such that none of \( a_i, b_i \) is zero and \( a_i \neq b_i \) for all \( i \), detect all single stuck-at faults on the inputs of \( G^* \). These two tests also detect single stuck-at faults on primary inputs \( x_1, \ldots, x_n \) since a single path is sensitized from each \( x_i \) to the output \( g \). Observing the second and the fourth tests from the test set \( T \) from Theorem 7.1:

<table>
<thead>
<tr>
<th>( x_0 )</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( \ldots )</th>
<th>( x_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>\ldots</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>\ldots</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>\ldots</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>( m - 1 )</td>
<td>( m - 1 )</td>
<td>\ldots</td>
<td>( m - 1 )</td>
</tr>
</tbody>
</table>
we see that the assignments for \( x_1, \ldots, x_n \) satisfy the requirements \( a_i, b_i \neq 0 \) and \( a_i \neq b_i \) for all \( i \in N \). Thus, the test set \( T \) detects all single stuck-at faults on primary inputs as well as on the inputs of \( G^* \).

So, by adding to the circuit an extra multiplication mod \( m \) gate with an observable output the number of tests sufficient to detect all single stuck-at faults is reducible to four.

### 7.5 Conclusion

In this chapter, we investigate the testability of logic circuits realizing \( m \)-valued functions in modulo \( m \) sum-of-products form, with \( m \) being a prime greater than two. We consider two aspects of the problem - the number of tests required for fault detection, and the generation of tests.

We prove that there exists a set of four tests detecting all single stuck-at faults on internal lines in the circuit. Furthermore, this set of tests is independent of the function being realized and therefore universal.

We propose two alternative techniques for the testing of primary inputs. The first one is to generate a test set of maximum length \( 2n \). We give a procedure for finding this test set and analyze its effectiveness. It is shown that the procedure effectively generates the tests for a primary input \( x_i \) when the circuit has a multiplication mod \( m \) gate depending on \( x_i \) and a small number \( k \) of other primary inputs. The smaller the \( k \), the easier it is to find the test for \( x_i \). We prove that for a random circuit implementing an \( m \)-valued \( n \)-variable function in modulo \( m \) sum-of-products form, the probability that \( k \leq 1 \) is greater than 99.99% for any \( x_i \), provided \( n \geq 3 \) and \( m \geq 3 \). Since this probability is very high, it is most likely that tests for primary inputs can be generated effectively using the proposed procedure.

The second technique for the testing of primary inputs we propose is to modify the circuit in such a way that the four tests for internal lines also detect all single stuck-at faults on primary inputs. We show that this can be accomplished by adding
to the circuit an extra multiplication mod $m$ gate with an observable output. The main advantage of this approach as compared to the first one is that the set of tests detecting all single stuck-at faults in the circuit is reduced to just four tests and, moreover, this set is universal and therefore no test generation procedure is required.
Chapter 8

Conclusion

This chapter summarizes the research contributions of this dissertation and discusses several areas for future work following from it.

This dissertation considers properties of Boolean and multiple-valued functions and uses them as a foundation for the development of several algorithms for logic synthesis of combinational logic circuits. Some other problems, related to logic synthesis, such as test generation for logic circuits and synthesis of easily testable circuits, are also addressed.

The main areas from the theory of Boolean and multiple-valued functions considered are: functional completeness, canonical forms, minimization of functions, discrete differences and functional decomposability. The primary contributions of the dissertation are the following:

- an efficient algorithm for three-level AND-OR-XOR minimization for Boolean functions and an upper bound on the number of products in the AND-OR-XOR expansion;

- a proof of the functional completeness of the set \{addition modulo \( m \), minimum\}, for \( m > 2 \), a constructive proof of the existence of a canonical form over this set and an algorithm for computing such a canonical form;
• a definition of a new multiple-valued discrete difference full sensitivity, an investigation of its usefulness to generate tests for multiple-valued logic circuits and a lower bound on the number of functions fully sensitive to their variables;

• an algorithm for generating the composition tree for any function fully sensitive to its variables;

• a proof that the logic circuits realizing modulo $m$ sum-of-product form, for $m > 2$, are easily testable, providing a technique for synthesis of multiple-valued logic circuits requiring just four universal tests for detection of all single stuck-at faults.

It is important to stress that the application of the theoretical results of this dissertation have been demonstrated only in so far as they are applicable to logic synthesis. This choice may give an incomplete idea of the actual realm of applications of these results. Since most of the theory is developed on a functional level, it might be of use in any area having functions as an object of study, such as combinatorial optimization or utility theory.

Several issues of this dissertation could be explored in more depth.

First, extending the theory of composition trees to handle heterogeneous functions seems to be a challenging problem. Such a theory would be of a great practical use, being a base for systematic logic synthesis of multi-level Boolean circuits. Another interesting open problem is how to extend the notion of composition tree to the multiple-output case. The algorithm for finding composition trees from Chapter 6 handles single-output problems only. Since most real-life problems are multiple-output, extending the algorithm to manage such cases is a topic for future work.

Second, since the algorithm AOXMIN developed in Chapter 4 shows a very good performance on benchmark functions, it is worth additional research development. Presently, it synthesizes a three-level AND-OR-XOR circuit, but it can easily be modified to generate multi-level circuits, by applying the algorithm subsequently to
the resulting AND-OR subcircuits.

Next, the algorithm for computing full sensitivity presented in Chapter 5 requires the expression of the function in a canonical form with no simplification allowed (i.e. $m^n$ terms for an $m$-valued $n$-variable function). Therefore, it becomes infeasible for large values of $m$ and $n$. Further work needs to be done to find a more efficient algorithm, using a more compact representation of functions, such as multiple-valued decision diagrams [40]. The same argument applies to the algorithm for constructing the new canonical form developed in Chapter 3, which uses the truth table coefficients of the function as its input.

And finally, Chapter 7 suggests a synthesis technique for multiple-valued logic circuits requiring just four tests for detection of all single stuck-at faults in the circuit. However, the circuit which is easiest for testing may not be the simplest possible one. Finding a trade-off between the simplicity of a circuit and its testability is an interesting topic for future work.
Appendix A

Current-Mode CMOS Multiple-Valued Circuits

This appendix describes a circuit technology called *multiple-valued current-mode CMOS*. This appendix is referred to in Chapter 3.

The multiple-valued current-mode CMOS circuit family is a relatively recent innovation. Most of the publications on multiple-valued CMOS logic circuits concern voltage-mode circuits. One reason for this choice is the fact that a MOS transistor is voltage-controlled. However, in 1986 Onneweer and Kerkhoff [45] have shown that most voltage-mode multiple-valued CMOS circuits are not competitive with two-valued circuits in terms of circuit complexity and propagation delay. They proposed using current-mode multiple-valued CMOS circuits instead and showed that these circuits can naturally handle *m*-valued signals whereas voltage-mode circuits become rather complicated for values of *m* greater than two. Various realizations of current-mode multiple-valued CMOS logic circuits have been discussed since then, and some threshold logic arithmetic circuits have been shown to be superior to the best corresponding two-valued ones at the same period [10], [11], [12], [33].

The structure of the appendix is as follows. In Section A.1, the basic elements of current-mode multiple-valued CMOS logic circuits are described. In Section A.2 the properties of three experimental circuits: MIN, MAX and addition modulo *m* are investigated by simulation (using the HSPICE program). The discussion is given in
terms of three-valued circuits, but the techniques are easily extended to circuits using a higher number of logic levels. A new addition modulo \( m \) circuit implementation, using 14 transistors only, is proposed and its properties are summarized. Section A.3 concludes the appendix.

A.1 Basic operations and symbols.

Current-mode circuits use analog current summing to create the algebraic weighted sum or difference of input currents. This function requires no passive or active components. The currents are usually defined to have logical levels that are integer multiples of a reference current unit. Currents can be copied, scaled, and algebraically sign-changed with a simple current mirror realized in any MOS technology. The sum or difference of currents is then usually decoded into the desired multiple-valued function by:

1. comparing it to multiple current thresholds using some form of current comparator, and

2. using comparator-controlled switches to direct properly scaled currents to the outputs

Different implementations of basic elements for current-mode circuits have been reported by several authors [12], [25], [31], [33], [45], [62], [63]. Below we give an overview and comparison of these implementations.

A.1.1 Constant-current source

A constant-current source is realized by a depletion-mode PMOS transistor with connected gate and source [31], [33]. In the ideal case, the saturation value of the drain current \( I_d \) used as a constant is written as

\[
I_d = K'(W/L)(V_T)^2
\]
where $K$ is the transconductance parameter, $W$ is the channel width, $L$ is the channel length, and $V_T$ is the threshold voltage of the depletion-mode PMOS transistor. This type of current source is quite insensitive to fluctuations of the supply voltage $V_{dd}$, and requires no connection other than $V_{dd}$. Figure A.1 shows the circuit configuration and the symbol for current source.

![Current Source](image)

Figure A.1: Current source: (a) circuit configuration, (b) symbol.

### A.1.2 Current mirror

In current-mode circuits a current mirror is used for two purposes. One is to invert the current direction, another is to produce replica of an input current. There are two types of current mirrors: N-type and P-type. These circuits are realized by enhancement-mode NMOS and PMOS transistors, respectively.

An N-type (P-type) mirror works as follows (see Figure A.2). Current $I_I$, forced externally to flow in (extracted from) the drain-gate connected transistor $T_0$, establishes a gate-source voltage at which each of $T_1, T_2, \ldots, T_n$ conduct the same current, extracting it from (inducing it into) connected circuits. The currents $I_{O1}, I_{O2}, \ldots, I_{On}$, produced at drains, are determined by input current $I_I$ and the ratio between output and input W/L values. The ratio is usually chosen so that:

$$I_{O_i} = \begin{cases} I_I & \text{if } I_I > 0 \\ 0 & \text{if } I_I \leq 0. \end{cases}$$
A.1.3 Current comparator

The classical current comparator as used by Current et al. in [12] and Yamakawa in [62] is shown in Figure A.3. The outputs of an NMOS current mirror are connected to the high-impedance control input of a switch. The voltage at this node is determined by the difference of the two input currents. The accuracy of this current comparator is limited by the finite output impedance of the current mirrors and by poor matching of the characteristics of N- and PMOS transistors.

Figure A.3: Classical current comparator.
A different current comparator with better characteristics was proposed by Onneweer and Kerkhoff [45] (Figure A.4). It consists of two NMOS current mirrors forming a positive-feedback loop. If the current mirrors are ideal, the gain of the feedback loop is unity. All signals in this circuit are currents. Its operation can be described as follows. Let \( T(x, y) \) is defined as:

\[
T(x, y) := \begin{cases} 
0 & \text{if } x < i \\
 x & \text{if } x > i 
\end{cases}
\]

Then \( IA'' = T(IA, IB) \) and \( IB'' = T(IB, IA) \).

![Figure A.4: Threshold current comparator of Onneweer and Kerkhoff.](image)

The comparator derives its switching nature from the positive feedback. It has two states, and in each state one of the output currents is zero while the other is a copy of the largest of the input currents. It can operate as a threshold switch by keeping \( IB \) constant, using \( IA \) as the (variable) input current. The threshold current is then equal to \( IB \), and the output current \( IB'' \) can assume the values 0 and \( IB \). By changing the channel width of the MOS transistors, the values of threshold and output currents can be changed. The switching speed of this circuit is determined by the charging time of the input nodes by the input currents, and therefore depends very much on the signal currents.

### A.2 Implementation of basic logic gates using current-mode CMOS circuits

We have simulated MIN, MAX and addition modulo 3 three-valued circuits with the HSPICE to explore the possibilities and properties of current-mode multiple-valued
APPENDIX A. CURRENT-MODE CMOS MULTIPLE-VALUED CIRCUITS

CMOS logic circuits. We chose the currents 0μA, 15μA, and 30μA to represent logic values 0, 1 and 2. A current direction is defined as positive when current is fed into the node and as negative, otherwise. $V_{dd}$ is chosen to be standard 5V. The PMOS and NMOS transistor parameters used in the simulation are presented in Table A.1. These are taken from the reference [45]. All devices have a channel length of 5μm and a channel width of 10μm.

Table A.1: HSPICE simulation parameters.

```
.MODEL N NMOS LEVEL=3 VT0=0.70 GAMMA=0.7 PHI=0.762
+ PB=0.74 CGSO=2.4E-10 CGDO=2.4E-10 CGBO=3.4E-10
+ CJ=3.46E-4 MJ=0.93 CJSW=7.61E-10 MJSW=0.28 JS=1.5E-5
+ TOX=25NM NSUB=4.1E16 NFS=1E10 TPG=1 XJ=0.61E-6
+ LD=.18E-6 VMAX=5E4 XQC=0.4 RSH=30
+ THETA=0.067 KAPPA=84.0E-6
*
.MODEL P PMOS LEVEL=3 VT0=-0.70 GAMMA=0.6 PHI=0.711
+ PB=0.87 CGSO=3.4E-10 CGDO=3.4E-10 CGBO=3.4E-10
+ CJ=7.12E-4 MJ=0.40 CJSW=9.42E-10 MJSW=0.29 JS=1E-6
+ TOX=25NM NSUB=1.5E16 NFS=1E10 TPG=-1 XJ=0.62E-6
+ LD=.25E-6 VMAX=5E4 XQC=0.4 RSH=95
+ THETA=0.17 KAPPA=34.0E-6
```

The choice of the unit current, the voltage $V_{dd}$ and the transistor's gate widths and lengths represents a trade-off between power dissipation and switching speed. Different authors uses different values for these parameters. For example, Onneweer and Kerkhoff [45] work with a unit current of 15μA, transistor sizes $L = 5\mu m$ and $W = 10\mu m$, and $V_{dd} = 3V$. Vranesic and Zilic [64] use a unit current of 20μA, $L = 3\mu m$ and $W = 6\mu m$, and $V_{dd} = 5V$. Kawahito et al. [33] use a unit current of 31μA with transistor sizes $L = 2.8\mu m$ and $W = 9\mu m$, and $V_{dd} = 5V$. For each of the circuits described, we conducted 6 experiments: for $V_{dd} = 3$ and $V_{dd} = 5$, and for unit currents of 10μA, 15μA and 20μA. The results show, that some of the circuits (e.g.
the threshold current comparator) do not operate correctly for $V_{dd} = 3V$. Decreasing the value of the unit current leads to decreasing of the power dissipation, but increases the possibility of misinterpreting the current levels. The best choice seems to be unit current $15\mu A$ and $V_{dd} = 5V$.

A.2.1 MIN and MAX circuits

In this section we compare two different implementations of MIN and MAX circuits. Both implementations are independent of the value of $m$, i.e. the number of transistors in them does not change with increasing $m$. The first one, shown on Figure A.5, is proposed by Onneweer and Kerkhoff [45]. Their circuit is based on the threshold current comparator (Figure A.4). If the two outputs of the comparator are summed, the maximum value of the two input currents $\text{max}(I_A, I_B)$ is obtained. The minimum value $\text{MIN}(I_A, I_B)$ is generated by subtracting the maximum value $\text{MAX}(I_A, I_B)$ from the arithmetic sum of $I_A$ and $I_B$. Current mirrors are included to distribute the input currents. The total number of transistors in this circuit is 16.

![Figure A.5: Minimum-maximum circuit of Onneweer and Kerkhoff.](image)

The second implementation was proposed by Zhijian and Hong [65] (Figure A.6a and b). Each MAX and MIN circuit consists of five transistors only. Their operations can be described as follows. Let the operation $\ominus$, called bounded difference operation be defined as:
\[
x \odot y := \begin{cases} 
  x - y & \text{if } x \geq y \\
  0 & \text{if } x < y.
\end{cases}
\]

where "-" is the regular arithmetic subtraction.

Then

\[
MAX(x, y) = x + (y \odot x) = y + (x \odot y)
\]

\[
MIN(x, y) = x - (x \odot y) = y - (y \odot x).
\]

where "-" and "+" are the regular arithmetic operations of subtraction and sum, correspondently.

![Figure A.6: (a) Minimum and (b) Maximum circuits of Zhijian and Hong.](image)

The operation of the MIN and MAX circuits is basically analog in nature. Therefore its accuracy is very important. The simulation results show that the MIN and MAX circuits of Zhijian and Hong have better characteristics than the MIN/MAX circuit of Onneweer and Kerkhoff. The MIN/MAX circuit of Onneweer and Kerkhoff (Figure A.7) has 25ns worst case delay time (measured between 10% and 90% of the signal), and power dissipation between 0.0016 and 1.38mW, while the MIN and MAX circuits of Zhijian and Hong (Figure A.8) have 20ns worst case delay time and power dissipation between 0 and 0.6mW.

### A.2.2 Addition modulo \(m\) circuit

In this section we compare two different implementations of a addition modulo \(m\) circuit. The first is a addition modulo 4 adder proposed by Zilic and Vranesic [64].
Figure A.7: The simulation results for the MIN/MAX circuit of Onneweer and Kerkhoff: (a) IA, (b) IB, (c) MIN(IA,IB), (d) MAX(IA,IB), (e) power dissipation.
Figure A.8: The simulation results for the MIN and MAX circuits of Zhijian and Hong: (a) IA, (b) IB, (c) MIN(IA,IB), (d) MAX(IA,IB), (e) power dissipation.
A simple observation was made that the addition modulo 4 differs from the absolute difference in only two entries of the truth table (see Table 2). Therefore the addition operation is realized as absolute difference plus a correction circuit for the two entries outlined in bold in Table 2.

<table>
<thead>
<tr>
<th>±</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AD</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2 Addition modulo 4 compared to absolute difference.

The implementation of the absolute difference circuit follows from the expression below:

\[ |A - B| = \text{ramp}(A - B) + \text{ramp}(B - A) \]

where \( \text{ramp}(x - y) \) is the operation defined as

\[ \text{ramp}(x - y) := \begin{cases} 
  x - y & \text{if } x \geq y \\
  0 & \text{if } x < y.
\end{cases} \]

where ",-" is the regular arithmetic subtraction.

Note the similarity between this operation and the operation \( \odot \) defined above. Figure A.9a shows the realization of the absolute difference circuit.

The correction factor involves adding a current of value 2 when the inputs \((A, B)\) have values \((1, 2)\) or \((2, 1)\). The correction circuit is shown on Figure A.9b.

This implementation of addition modulo \( m \) circuit is dependent on the value of \( m \), since for different \( m \) the different correction circuit has to be built. Furthermore, the total number of transistors in this circuit is 24. Instead, we propose an addition modulo \( m \) circuit implementation, independent of the value of \( m \) and consisting of 14 transistors only. The design for the general \( m \)-valued case is shown on Figure A.10.
Figure A.9: (a) Absolute difference circuit (b) Correction circuit.
The circuit is based on the threshold current comparator (Figure A.4). The arithmetic sum of two input currents $I_A$ and $I_B$ is compared with the current which has value $m$ (current units), generated by current source $CS_1$. The output of the current comparator is the current \( I_{\text{comp}} = T(m, I_A + I_B) \), i.e.:

\[
I_{\text{comp}} = T(m, A + B) = \begin{cases} 
0 & \text{if } m < I_A + I_B \\
 m & \text{if } m > I_A + I_B 
\end{cases}
\]

If the sum $I_A + I_B + 0.5$ is less than $m$ (current units), then the output current of the current comparator $I_{\text{comp}}$ is $m$ (current units). This current "compensates" the $m$ current units, extracted by the current source $CS_2$, and the current $I'$ becomes equal to zero. Thus, the output current of the circuit for this case is $IS = I_A + I_B - 0 = I_A + I_B$.

If the sum $I_A + I_B + 0.5$ is greater than $m$ (current units), then the output current of the current comparator $I_{\text{comp}}$ is 0. In this case $I'$ becomes equal to $m$.
APPENDIX A. CURRENT-MODE CMOS MULTIPLE-VALUED CIRCUITS

current units, and these $m$ current units are extracted from the arithmetic sum of $I_A$ and $I_B$. Thus, the output of the circuit is $I_S = I_A + I_B - I' = I_A + I_B - m$.

The simulation results for this implementation of the addition modulo $m$ circuit for $m = 3$ are shown on Figure A.11. Worst case delay time is 25ns. Power dissipation varies between 0 and 1.40mW. The delay time and power dissipation of the addition modulo $m$ circuit of Zilic and Vranesic [64] are not reported, so we cannot make a comparison.

A.3 Conclusion

This appendix presents a multiple-valued current-mode CMOS circuit family. We show the current-mode CMOS implementation of MIN, MAX and addition modulo $m$ circuits and investigate their properties by simulation using the HSPICE program. We propose a new addition modulo $m$ circuit implementation, independent of the value of $m$ and consisting of 14 transistors only.

Compared to voltage-mode circuits, current-mode CMOS circuits do not need passive devices (resistors) to establish some of the output levels and use one power supply line only. Furthermore, the linear summation in current-mode CMOS circuits is performed by simple wiring, which reduces the interconnection complexity and decreases the area of the resulting circuit. However, since linear summation of currents is analog in nature and not level-restoring, the main problem with current-mode CMOS circuits are the noise margins. With the increase in of the number of current levels, the noise margins constraints are more and more difficult to satisfy. The current-mode CMOS circuits with three or four current levels seem to be the most efficient ones.
Figure A.11: The simulation results for the addition modulo $m$ circuit: (a) $I_A$, (b) $I_B$, (c) $\text{sum}(I_A,I_B)$, (d) power dissipation.
Bibliography


