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)

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. 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)

Contents

Contents iii

List of Tables vi

List of Figures vii

Acknowledgment ix

Dedication x

1 Introduction 1

2 Background 7
2.1 Notation 7
2.2 Basic notions 8
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 $\{\dot{+}, \cdot\}$ 17
3.4 Properties of the operations of the algebra $\mathcal{B}$ 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

4.4.1 Dividing the cubes into equivalence classes 42
4.4.2 Obtaining T_1 and T_2 45
4.4.3 Constructing *g1_init* and *g2_init* 46
4.4.4 Determining common don't cares in *g1_init* and *g2_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

Bibliography **122**

List of Tables

4.1 Benchmark results. 49
4.2 AOXMIN results for *t*481. 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 ith row of the matrix $\mathbf{F}^2$ by the matrix $\mathbf{T}^2$. 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 *g1_init* and *g2_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 $A_1, A_2, \ldots, A_n$. 66
5.5 Plots for $\epsilon(m, n)$ as a function of n for fixed $m = 3, 5$ and 10. 68
5.6 Plots for $\epsilon(3, n)$ and $\epsilon^*(3, n)$ as functions of n for $m = 3$. 70

6.1 Simple disjunctive decomposition 74
6.2 Decomposition chart for an example function in Σ 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 an 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

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

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

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

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

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 Σ, 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 "$'$" 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, etc to denote elements over M, and lower-case letters f, g, h, g_1, g_2, etc to denote functions. We use $x_1, x_2, \ldots, x_n$ to denote variables of the functions, and use $N = \{1, 2, \ldots, n\}$ to denote the set of indices of these variables. We use capital letters A, B, C, 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 **A**, **B**, **C**, etc to denote matrices.

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* the set 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* $f : A \to B$ 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 $f : A \to B$ 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.

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 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* $\bullet$ 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 $c = a \bullet b$ 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* $\mathcal{A} = \langle M; J, +, \cdot : 0, m-1 \rangle$, *where*

(*i*) $M := \{0, 1, \ldots, m-1\}$ *is the totally ordered carrier of* $\mathcal{A}$;

(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* $\overset{i}{x}$*:*

(iii) " $+$ " *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 " $-$ " is the usual arithmetic subtraction.

Functional completeness of $\mathcal{A}$ 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 $\mathcal{A}$. 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* $\mathcal{A}$ *of type*

$$f(x_1, \ldots, x_n) = \sum_{i=0}^{m^n-1} c_i \; \overset{i_1}{x}_1 \; \overset{i_2}{x}_2 \ldots \; \overset{i_n}{x}_n,$$

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, $\overset{0}{x} = x'$ and $\overset{1}{x} = x$. 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:

$$f(\underline{x}) = \sum_{j \in M} \overset{j}{x_i} \, f(\underline{x}_i^j) \tag{2.1}$$

where $\underline{x} := (x_1, \ldots, x_n)$ and $\underline{x}_i^j$ is the vector $\underline{x}$ with $x_i = j$, i.e.

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

with $j \in M$, $i \in N$. So, in terms of these notations, $f(\underline{x}_i^j)$ denotes the subfunction of the function $f(\underline{x})$ 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-

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:

$$f(x_1, \ldots, x_n) = \sum_{i=0}^{2^n-1} c_i \; x_1^{i_1} \; x_2^{i_2} \ldots \; x_n^{i_n}, \tag{3.1}$$

where the sign $\sum$ stands for XOR. $c_i \in \{0, 1\}$ are constants, $(i_1 i_2 \ldots i_n)$ 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:

$$f(x_1, \ldots, x_n) = \sum_{i=0}^{2^n-1} c_i \; {}^{k_1}x_1^{i_1} \; {}^{k_2}x_2^{i_2} \ldots \; {}^{k_n}x_n^{i_n} \tag{3.2}$$

where $c_i \in \{0, 1\}$ are constants, $(i_1 i_2 \ldots i_n)$ and $(k_1 k_2 \ldots k_n)$ 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: ${}^0x_j^1 = x_j$, ${}^1x_j^1 = x_j'$ and, for any k_j, ${}^{k_j}x_j^0 = 1$. When k is fixed, this form is unique for a given function.

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:

$$f(x_1, \ldots, x_n) = \sum_{i=0}^{m^n-1} c_i \; x_1^{i_1} \; x_2^{i_2} \ldots \; x_n^{i_n}. \tag{3.3}$$

where the sign $\sum$ stands for multiplication modulo m, $c_i \in M$ are constants, $(i_1 i_2 \ldots i_n)$ is the m-ary expansion of i with i_1 being the least significant digit, and the term $x_j^{i_j}$ denotes the i_jth 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:

$$f(x_1, \ldots, x_n) = \sum_{i=0}^{m^n-1} c_i \; {}^{i_1}x_1^{k_1} \; {}^{i_2}x_2^{k_2} \ldots \; {}^{i_n}x_n^{k_n} \tag{3.4}$$

where $c_i \in M$ are constants, $(i_1 i_2 \ldots i_n)$ and $(k_1 k_2 \ldots k_n)$ 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 $x_j^{i_j \oplus k_j}$ 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:

$$f(x_1 + k_1, x_2 + k_2, \ldots, x_n + k_n) = \sum_{i=0}^{m^n-1} c_i \; x_1^{i_1} \; x_2^{i_2} \ldots \; x_n^{i_n} \tag{3.5}$$

where $c_i \in M$ are constants. $(i_1 i_2 \ldots i_n)$ and $(k_1 k_2 \ldots k_n)$ 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 $\mathcal{B}$ defined as follows:

Definition 3.1 *A multiple-valued algebra $\mathcal{B}$ is an algebra $\mathcal{B} = \langle M; \dot{-}, \cdot, J: 0, m-1 \rangle$, where*

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

(ii) "$\dot{-}$" 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 "$\dot{-}$" 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 "$\dot{-}$". Recall that, for convenience, we write $J_i x$ as $\overset{i}{x}$.

Every element a of M has an *inverse* $-a$ (with respect to the "$\dot{-}$" operation), defined as

$$-a := \underbrace{a \dot{-} a \dot{-} \ldots \dot{-} a}_{m-1 \text{ times}}.$$

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*

$$x' := (m-1) \dot{-} (-x)$$

Obviously $x \dot{+} x' = m-1$ since for any x ranging in M, $x \dot{-} (-x) = 0$.

Definition 3.3 *Subtraction modulo m "$\dot{-}$" is defined by*

$$x \dot{-} y := x \dot{-} (-y)$$

where x and y denote multiple-valued variables.

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

The chain-based Post algebra (Definition 2.1), based on the operations MIN, MAX and the set of all literal operators, is well-known to be functionally complete with constants [44]. Since MAX can be expressed through 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 $\{\dot{-}, \cdot, J\}$ 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 $\{\dot{-}, \cdot\}$. 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 "$\dot{-}$" and "$\cdot$" and the constant $(m-1)$, which proves the functional completeness (with constants) of $\{\dot{-}, \cdot\}$.

Property 3.4 *The literal $\overset{i}{x}$ can be expressed in terms of "$\dot{-}$", "$\cdot$" and $'$ as follows:*

$$\overset{i}{x} = (g'(x,i) \dot{-} g(x,i) \cdot g'(x,i))'.$$

where $g(x,i) := (x \dot{-} i') \cdot (x' \dot{-} i)$.

Proof: 1) Let $x = i$. Then $\overset{i}{x} = m-1$. On the other hand:

$$\begin{aligned} g(i,i) &= (i \dot{-} i') \cdot (i' \dot{-} i)) && \{\text{definition of } g(x,i)\} \\ &= (m-1) \cdot (m-1) && \{\forall a \in M : a \dot{-} a' = m-1\} \\ &= (m-1) && \{\text{idempotency of } \cdot\} \end{aligned}$$

Therefore

$$\begin{aligned} (g'(i,i) \dot{-} g(i,i)\ g'(i,i))' &= ((m-1)' \dot{-} (m-1) \cdot (m-1)')' && \{g(i,i) = m-1\} \\ &= (0 \dot{-} (m-1) \cdot 0)' && \{\text{Definition 3.2}\} \\ &= (0 \dot{-} 0)' && \{0 \text{ is the null element of } \cdot\} \\ &= 0' && \{0 \text{ is the unit element of } \dot{-}\} \\ &= m-1 && \{\text{Definition 3.2}\} \end{aligned}$$

Hence, for $x = i$, $\overset{i}{x} = (g'(x,i) \dot{-} g(x,i) \cdot g'(x,i))'$.

2) Let $x \neq i$. Then $\overset{i}{x} = 0$. 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

$$\forall x, i, x \neq i\ [(x \oplus i' \geq \lfloor \frac{m}{2} \rfloor) \Rightarrow (x' \oplus i < \lfloor \frac{m}{2} \rfloor)].$$

1)	$x \oplus i' \geq \lfloor \frac{m}{2} \rfloor$	{hypothesis}
2)	$\lfloor \frac{m}{2} \rfloor \leq x \oplus i' < m-1$	$\{(x \neq i) \Rightarrow (x \oplus i' \neq m-1)\}$
3)	$\exists z\ [x \oplus i' \oplus z = m-1]$	$\{\langle M, \oplus \rangle$ is a group$\}$
4)	$1 \leq z \leq \lfloor \frac{m}{2} \rfloor$	{(2).(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).(6)}
8)	$x' \oplus i = z \ominus 1$	{Definition 3.3. $-1 = m-1$}
9)	$x' \oplus i < \lfloor \frac{m}{2} \rfloor$	{(4).(8)}

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

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

$$\begin{aligned} (a' \oplus aa')' &= (a' \oplus a)' \quad \{(a < \lfloor \tfrac{m}{2} \rfloor) \Rightarrow (aa' = a)\} \\ &= (m-1)' \quad \{\forall a \in M : a \oplus a' = m-1\} \\ &= 0 \qquad \{\text{Definition 3.2}\} \end{aligned}$$

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

□

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 $\mathcal{B}$. 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 $\mathcal{B}$. These properties are presented and proved in the next section.

3.4 Properties of the operations of the algebra $\mathcal{B}$

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) $(\overset{i}{x})' = \sum_{j \in M-\{i\}} \overset{j}{x}$

b) $f \cdot \overset{i}{x} = f \oplus [(-f) \cdot (\overset{i}{x})']$

c) $f \cdot \overset{i}{x} + g \cdot \overset{j}{x} = f \cdot \overset{i}{x} \oplus g \cdot \overset{j}{x}$

d) $f \cdot (\overset{i}{x} \oplus \overset{j}{x}) = f \cdot \overset{i}{x} \oplus f \cdot \overset{j}{x}$

e) $\overset{i}{x} \cdot (f \oplus g) = \overset{i}{x} \cdot f \oplus \overset{i}{x} \cdot g$

f) $\overset{i}{x} \cdot (f \odot g) = \overset{i}{x} \cdot f \odot \overset{i}{x} \cdot g$

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

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

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

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

$$\begin{aligned} f \oplus [(-f) \cdot (\overset{i}{x})'] &= f \oplus [(-f) \cdot (m-1)'] \\ &= f \oplus [(-f) \cdot 0)] \\ &= f \oplus 0 \\ &= f \end{aligned}$$

2) Let $x \neq i$. Then $f \cdot \overset{i}{x} = 0$. On the other hand:

$$\begin{aligned} f \dot{+} [(-f) \cdot (\overset{i}{x})'] &= f \dot{+} [(-f) \cdot 0'] \\ &= f \dot{+} [(-f) \cdot (m-1)] \\ &= f \dot{-} f \\ &= 0 \end{aligned}$$

Hence for both cases $f \cdot \overset{i}{x} = f \dot{+} [(-f) \cdot (\overset{i}{x})']$

(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 $f \cdot \overset{i}{x} + g \cdot \overset{j}{x} = f \cdot 0 + g \cdot \overset{j}{x} = 0 + g \cdot \overset{j}{x} = g \cdot \overset{j}{x}$, and the right hand side is $f \cdot \overset{i}{x} \dot{-} g \cdot \overset{j}{x} = f \cdot 0 \dot{-} g \cdot \overset{j}{x} = 0 \dot{-} g \cdot \overset{j}{x} = g \cdot \overset{j}{x}$.

Hence for $x \neq i$, $f \cdot \overset{i}{x} + g \cdot \overset{j}{x} = f \cdot \overset{i}{x} \dot{-} g \cdot \overset{j}{x}$. 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 $f(\overset{i}{x} \dot{+} \overset{j}{x}) = f \cdot (0 \dot{+} \overset{j}{x}) = f \cdot \overset{j}{x}$, and the right hand side is $f \cdot \overset{i}{x} \dot{-} f \cdot \overset{j}{x} = f \cdot 0 \dot{-} f \cdot \overset{j}{x} = f \cdot \overset{j}{x}$.

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

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

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

Hence for both cases $\overset{i}{x} \cdot (f \dot{+} g) = \overset{i}{x} \cdot f \dot{+} \overset{i}{x} \cdot g$.

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

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

Hence for both cases $\overset{i}{x} \cdot (f \hat{-} g) = \overset{i}{x} \cdot f \hat{-} \overset{i}{x} \cdot g$.

□

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 $f(\underline{x}_i^j)$ denotes a subfunction of the function $f(\underline{x})$ with the variable x_i being fixed to the value j, i.e. $f(\underline{x}_i^j) = f(x_1, \ldots, x_{i-1}, j, x_{i+1}, \ldots, x_n)$. Then, the positive and negative decompositions of Boolean functions are of form [36]:

$$\begin{aligned} f(\underline{x}) &= f(\underline{x}_n^0) \hat{-} x_n (f(\underline{x}_n^0) \hat{-} f(\underline{x}_n^1)) && \text{positive decomposition} \\ &= f(\underline{x}_n^1) \hat{-} x_n' (f(\underline{x}_n^0) \hat{-} f(\underline{x}_n^1)) && \text{negative decomposition} \end{aligned} \tag{3.6}$$

Theorem 3.6 is the general decomposition theorem for a function $f(\underline{x})$ 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 $f(\underline{x})$ can be decomposed with respect to the variable x_n and a given $i \in M$ in the following way:*

$$f(\underline{x}) = f(\underline{x}_n^i) \hat{-} \sum_{j=1}^{m-1} (f(\underline{x}_n^{i \hat{\oplus} j}) \hat{-} f(\underline{x}_n^i)) \overset{i \hat{\oplus} j}{x_n}$$

Proof:

Using generalized Shannon decomposition (2.1) we can express the function $f(\underline{x})$ as follows:

$$
\begin{aligned}
f(\underline{x}) &= \overset{0}{x}_n\, f(\underline{x}_n^0) + \overset{1}{x}_n\, f(\underline{x}_n^1) + \ldots + \overset{m-1}{x}_n\, f(\underline{x}_n^{m-1}) && \{(2.1)\}\\
&= \overset{0}{x}_n\, f(\underline{x}_n^0) \oplus \overset{1}{x}_n\, f(\underline{x}_n^1) \oplus \ldots \oplus \overset{m-1}{x}_n\, f(\underline{x}_n^{m-1}) && \{\text{Property 3.5(c)}\}\\
&= (f(\underline{x}_n^0) \oplus [-f(\underline{x}_n^0) \cdot (\overset{0}{x}_n)']) \oplus \sum_{j=1}^{m-1} \overset{j}{x}_n\, f(\underline{x}_n^j) && \{\text{Property 3.5(b)}\}\\
&= (f(\underline{x}_n^0) \oplus [-f(\underline{x}_n^0) \cdot \sum_{i=1}^{m-1} \overset{i}{x}_n]) \oplus \sum_{j=1}^{m-1} \overset{j}{x}_n\, f(\underline{x}_n^j) && \{\text{Property 3.5(a)}\}\\
&= f(\underline{x}_n^0) \oplus \sum_{j=1}^{m-1} (-f(\underline{x}_n^0))\, \overset{j}{x}_n \oplus \sum_{j=1}^{m-1} \overset{j}{x}_n\, f(\underline{x}_n^j) && \{\text{Property 3.5(d)}\}\\
&= f(\underline{x}_n^0) \oplus \sum_{j=1}^{m-1} f(\underline{x}_n^j)\, \overset{j}{x}_n \oplus (-f(\underline{x}_n^0))\, \overset{j}{x}_n && \{\text{commutativity of } \oplus\}\\
&= f(\underline{x}_n^0) \oplus \sum_{j=1}^{m-1} (f(\underline{x}_n^j) \oplus (-f(\underline{x}_n^0)))\, \overset{j}{x}_n && \{\text{Property 3.5(e)}\}\\
&= f(\underline{x}_n^0) \oplus \sum_{j=1}^{m-1} (f(\underline{x}_n^j) \ominus f(\underline{x}_n^0))\, \overset{j}{x}_n && \{\text{Definition 3.3}\}
\end{aligned}
$$

In the above derivation we expanded $\overset{0}{x}_n \cdot f(\underline{x}_n^0)$ using Property 3.5(b). If alternatively we expanded $\overset{i}{x}_n \cdot f(\underline{x}_n^i)$ for some $i \neq 0$. then the derivation gives the proof for the corresponding value of i.

□

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:

$$f(x_1, x_2) = f(\underline{x}_2^i) \oplus [(f(\underline{x}_2^{i\oplus 1}) \ominus f(\underline{x}_2^i))\, \overset{i\oplus 1}{x}_2] \oplus [(f(\underline{x}_2^{i\oplus 2}) \ominus f(\underline{x}_2^i))\, \overset{i\oplus 2}{x}_2].$$

The decompositions for all three possible values of i are:

For $i = 0$: $f(x_1, x_2) = f(\underline{x}_2^0) \oplus [(f(\underline{x}_2^1) \ominus f(\underline{x}_2^0))\, \overset{1}{x}_2] \oplus [(f(\underline{x}_2^2) \ominus f(\underline{x}_2^0))\, \overset{2}{x}_2]$.

For $i = 1$: $f(x_1, x_2) = f(\underline{x}_2^1) \oplus [(f(\underline{x}_2^2) \ominus f(\underline{x}_2^1))\, \overset{2}{x}_2] \oplus [(f(\underline{x}_2^0) \ominus f(\underline{x}_2^1))\, \overset{0}{x}_2]$.

For $i = 2$: $f(x_1, x_2) = f(\underline{x}_2^2) \oplus [(f(\underline{x}_2^0) \ominus f(\underline{x}_2^2))\, \overset{0}{x}_2] \oplus [(f(\underline{x}_2^1) \ominus f(\underline{x}_2^2))\, \overset{1}{x}_2]$.

For example, suppose $f(x_1, x_2)$ is defined by the table below:

		x_2		
		0	1	2
	0	0	1	2
x_1	1	1	1	2
	2	2	0	0

Then we have

x_1	$f(\underline{x}_2^0)$	$f(\underline{x}_2^1)$	$f(\underline{x}_2^2)$
0	0	1	2
1	1	1	2
2	2	0	0

and, for the decomposition with respect to x_2 with $i = 0$

$$\begin{aligned} f(x_1, x_2) &= f(\underline{x}_2^0) \dot{-} [(f(\underline{x}_2^1) \dot{-} f(\underline{x}_2^0)) \overset{1}{x}_2] \dot{-} [(f(\underline{x}_2^2) \dot{-} f(\underline{x}_2^0)) \overset{2}{x}_2] \\ &= f(\underline{x}_2^0) \dot{-} g_1(x_1) \overset{1}{x}_2 \dot{-} g_2(x_1) \overset{2}{x}_2 \end{aligned}$$

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

x_1	$g_1(x_1)$	$g_2(x_1)$
0	1	2
1	0	1
2	1	1

Obviously, if each of the subfunctions $f(\underline{x}_n^j)$, $j \in M$, in the decomposition of $f(\underline{x})$ is successively decomposed about the remaining variables, we finally get an expression in which $f(\underline{x})$ 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 $f(\underline{x})$ 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 $\overset{k_i}{x}_i$, 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 $\overset{0}{x}_1$ and $\overset{2}{x}_1$ in the canonical form, x_2 by literals $\overset{0}{x}_2$ and $\overset{1}{x}_2$, and x_3 by literals $\overset{1}{x}_3$ and $\overset{2}{x}_3$. 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 $[c_0\ c_1 \ldots c_{m^n-1}]$, $c_j \in M$. The notation ${}^i x^j$ used in the theorem below is defined as follows:

$${}^i x^j := \begin{cases} m-1 & \text{if } i = 0 \\ \overset{i \ominus j}{x} & \text{otherwise} \end{cases}$$

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:*

$$f(x_1, \ldots, x_n) = \sum_{j=0}^{m^n-1} c_j \, {}^{j_1}x_1^{k_1} \, {}^{j_2}x_2^{k_2} \ldots {}^{j_n}x_n^{k_n}$$

where $c_j \in M$ *are constants, and* $(j_n \ldots j_2 j_1)$ *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:

$$\begin{aligned}
f(x) &= f_i \ominus \sum_{j=1}^{m-1} (f_{i \ominus j} \ominus f_i) \, \overset{i \ominus j}{x} && \{\text{Theorem 3.6}\} \\
&= c_0 \ominus \sum_{j=1}^{m-1} c_j \, \overset{i \ominus j}{x} && \{\text{where } c_0 = f_i \text{ and } c_j = f_{i \ominus j} \ominus f_i\} \\
&= c_0 \, {}^0x^i \ominus \sum_{j=1}^{m-1} c_j \, {}^jx^i && \{{}^0x^i = m - 1, \text{and } {}^jx^i = \overset{i \ominus j}{x}, \text{for } j \neq 0\} \\
&= \sum_{j=0}^{m-1} c_j \, {}^jx^i
\end{aligned}$$

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 the variable x_{n+1} and a given $i \in M$ as:

$$f(x_1, \ldots, x_{n+1}) = f(\underline{x}^i_{n+1}) \ominus \sum_{p=1}^{m-1} (f(\underline{x}^{i \ominus p}_{n+1}) \ominus f(\underline{x}^i_{n+1})) \, \overset{i \ominus p}{x}_{n+1}$$

By the induction hypothesis, which assumes the result for the functions of n variables, we can express each of the subfunctions $f(\underline{x}^i_{n+1})$, $f(\underline{x}^{i \ominus p}_{n+1})$, $p \in M$ in the canonical form for some polarity $k = (k_n \ldots k_2 k_1)$. We use the notation c^r_j to denote the jth coefficient of the canonical form of the subfunction $f(\underline{x}^r_{n+1})$. Then we have:

$$f(x_1,\ldots,x_{n+1}) = \sum_{j=0}^{m^n-1} c_j^i\ {}^{j_1}x_1^{k_1}\ldots{}^{j_n}x_n^{k_n}$$
$$\oplus \sum_{p=1}^{m-1}\left(\sum_{j=0}^{m^n-1} c_j^{i\oplus p}\ {}^{j_1}x_1^{k_1}\ldots{}^{j_n}x_n^{k_n} \ominus \sum_{j=0}^{m^n-1} c_j^i\ {}^{j_1}x_1^{k_1}\ldots{}^{j_n}x_n^{k_n}\right)^{i\oplus p} x_{n+1}$$

To simplify the exposition, we use the notation ${}^jX^k$ to stand for the term ${}^{j_1}x_1^{k_1}\ldots{}^{j_n}x_n^{k_n}$. Then the above expression becomes:

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

{Definition 3.3}

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

{distributivity of " $-$ " over "$\ominus$"}

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

{commutativity of "$\ominus$"}

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

{Definition 3.3}

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

{Property 3.5(f)}

$$= \sum_{j=0}^{m^n-1} c_j^i\ {}^jX^k \oplus \sum_{p=1}^{m-1}\sum_{j=0}^{m^n-1}(c_j^{i\oplus p} \ominus c_j^i)\ {}^jX^k\ {}^{i\oplus p}x_{n+1}$$

{Property 3.5(f)}

$$= \sum_{j=0}^{m^n-1} c_j^i\ {}^jX^k\ {}^0x_{n+1}^i \oplus \sum_{p=1}^{m-1}\sum_{j=0}^{m^n-1}(c_j^{i\oplus p} \ominus c_j^i)\ {}^jX^k\ {}^px_{n+1}^i$$

$\{{}^0x^i = m-1, \text{and } {}^px^i = {}^{i\oplus p}x, \text{for } p\neq 0\}$

$$= \sum_{j=0}^{m^n-1} c_j^i \; {}^{j}X^{k\,0}x_{n+1}^i \oplus \sum_{j=0}^{m^n-1} (c_j^{i\dot{\ominus}1} \dot{-} c_j^i) \; {}^{j}X^{k\,1}x_{n+1}^i \oplus .. \oplus \sum_{j=0}^{m^n-1} (c_j^{i\dot{\ominus}(m-1)} \dot{-} c_j^i) \; {}^{j}X^{k\,m-1}x_{n+1}^i$$

{reordering}

$$= \sum_{j=0}^{m^{n+1}-1} c_j \; {}^{j}X^{k\;j_{n+1}}x_{n+1}^i.$$

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

□

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:

$$f(x_1, x_2) = \sum_{j=0}^{8} c_j \; {}^{j_1}x_1^{k_1} \; {}^{j_2}x_2^{k_2} = c_0 \dot{-} c_1 \; {}^{1\dot{-}k_1}x_1 \dot{-} c_2 \; {}^{2\dot{-}k_1}x_1 \dot{-} c_3 \; {}^{1\dot{-}k_2}x_2 \dot{-}$$
$$\dot{-} c_4 \; {}^{1\dot{-}k_1}x_1 \; {}^{1\dot{-}k_2}x_2 \dot{-} c_5 \; {}^{2\dot{-}k_1}x_1 \; {}^{1\dot{-}k_2}x_2 \dot{-} c_6 \; {}^{2\dot{-}k_2}x_2 \dot{-} c_7 \; {}^{1\dot{-}k_1}x_1 \; {}^{2\dot{-}k_2}x_2 \dot{-} c_8 \; {}^{2\dot{-}k_1}x_1 \; {}^{2\dot{-}k_2}x_2$$

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:

$$f(x_1, x_2) = c_0 \dot{-} c_1 \; {}^{2}x_1 \dot{-} c_2 \; {}^{0}x_1 \dot{-} c_3 \; {}^{1}x_2 \dot{-} c_4 \; {}^{2}x_1 \, {}^{1}x_2 \dot{-} c_5 \; {}^{0}x_1 \, {}^{1}x_2 \dot{-} c_6 \; {}^{2}x_2 \dot{-} c_7 \; {}^{2}x_1 \, {}^{2}x_2 \dot{-} c_8 \; {}^{0}x_1 \, {}^{2}x_2 \, .$$

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 kth 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 a s, 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 $\mathbf{F}^n$, defined next.

Let $\mathbf{F}_u^p$ 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 $[\mathbf{F}_u^p]_{ij}$ used in the definition refers to the submatrix in the ith row and jth column of the matrix $\mathbf{F}_u^p$.

Definition 3.8 *The $m^n \times m^n$ matrix $\mathbf{F}^n$ is defined as $\mathbf{F}_0^n$, where $\mathbf{F}_u^p$ is defined inductively by:*

1. $\mathbf{F}_u^0 := [f_u]$
2. $[\mathbf{F}_u^p]_{ij} := \mathbf{F}_{um+(i\ominus j)}^{p-1}$

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 $\mathbf{F}^2$ is constructed as follows:

1. For $u \in \{0,1,2\}$, $\mathbf{F}_u^0 := [f_u]$.

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

So:

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

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. $\mathbf{T}^n := \begin{bmatrix} \mathbf{T}^{n-1} & (m-1) \ddot{\cdot} \mathbf{T}^{n-1} & (m-1) \ddot{\cdot} \mathbf{T}^{n-1} & \dots & (m-1) \ddot{\cdot} \mathbf{T}^{n-1} \\ 0 & \mathbf{T}^{n-1} & 0 & \dots & 0 \\ 0 & 0 & \mathbf{T}^{n-1} & \dots & 0 \\ & & & \dots & \\ 0 & 0 & 0 & \dots & \mathbf{T}^{n-1} \end{bmatrix}$

where $\ddot{\cdot}$ 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:

$$\mathbf{T}^0 = [1], \qquad \mathbf{T}^1 = \begin{bmatrix} 122 \\ 010 \\ 001 \end{bmatrix}, \qquad \mathbf{T}^2 = \begin{bmatrix} 122 & 211 & 211 \\ 010 & 020 & 020 \\ 001 & 002 & 002 \\ 000 & 122 & 000 \\ 000 & 010 & 000 \\ 000 & 001 & 000 \\ 000 & 000 & 122 \\ 000 & 000 & 010 \\ 000 & 000 & 001 \end{bmatrix}.$$

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

$$\mathbf{C} = \mathbf{F}^n \circledcirc \mathbf{T}^n$$

Every row k of the resulting $m^n \times m^n$ matrix $\mathbf{C}$ 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 $\mathbf{F}^n \odot \mathbf{T}^n$ 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 $\mathbf{T}^n$ a "fast" procedure for performing $\mathbf{F}^n \odot \mathbf{T}^n$ is possible, with the total number of summations reduced to $n \times \frac{m-1}{m} \times m^n \times m^n$, 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 ith row of $\mathbf{F}^2$ by $\mathbf{T}^2$. The result is a vector of coefficients $[c_0 c_1 \ldots c_8]$ with polarity i. The total number of summations required is $n \times \frac{m-1}{m} \times m^n = 12$. The complete diagram for multiplication of $\mathbf{F}^2$ by $\mathbf{T}^2$ consists of $m^n = 9$ such fragments.

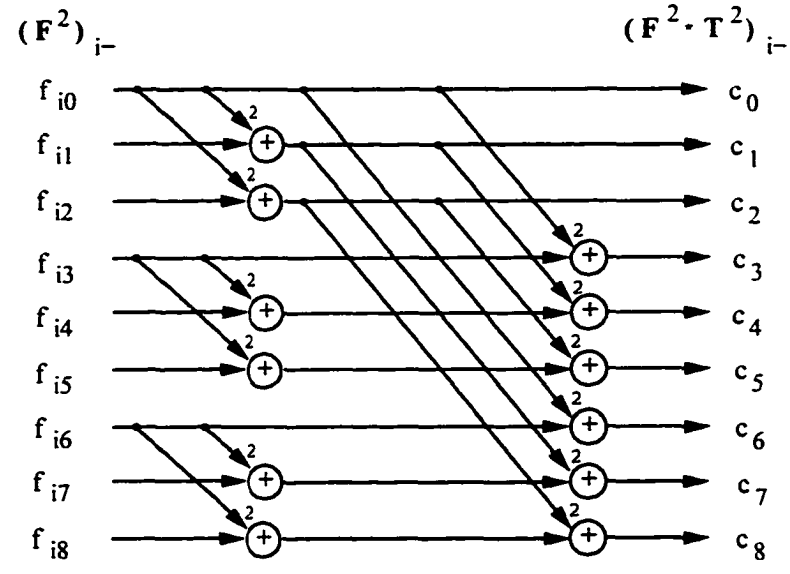


Figure 3.1: Diagram illustrating the multiplication of ith row of the matrix $\mathbf{F}^2$ by the matrix $\mathbf{T}^2$.

The following example illustrates the calculation of the matrix of coefficients $\mathbf{C}$.

Example 3.10. Consider the following 3-valued 2-variable function:

$$f(x_1,x_2) = 1\overset{0}{x}_1\overset{0}{x}_2 + 1\overset{1}{x}_1\overset{0}{x}_2 + \overset{2}{x}_1\overset{0}{x}_2 + 1\overset{1}{x}_1\overset{1}{x}_2 + \overset{1}{x}_1\overset{2}{x}_2 + 1\overset{2}{x}_1\overset{2}{x}_2 .$$

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

		x_2		
		0	1	2
	0	1	0	0
x_1	1	1	1	2
	2	2	0	1

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

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

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:

$$\begin{aligned} f(x_1,x_2) &= c_1\overset{1\dot{-}k_1}{x}_1 \dot{-} c_2\overset{2\ominus k_1}{x}_1 \dot{-} c_3\overset{1\ominus k_2}{x}_2 \dot{-} c_4\overset{1\ominus k_1}{x}_1\overset{1\ominus k_2}{x}_2 \dot{-} c_5\overset{2\ominus k_1}{x}_1\overset{1\ominus k_2}{x}_2 \dot{-} c_6\overset{2\ominus k_2}{x}_2 \dot{-} \\ &\quad \dot{-} c_7\overset{1\ominus k_1}{x}_1\overset{2\ominus k_2}{x}_2 \dot{-} c_8\overset{2\ominus k_1}{x}_1\overset{2\ominus k_2}{x}_2 \\ &= 0 \ominus 2\overset{1\ominus 0}{x}_1 \ominus 1\overset{2\ominus 0}{x}_1 \dot{-} 1\overset{1\ominus 2}{x}_2 \ominus 1\overset{1\ominus 0}{x}_1\overset{1\ominus 2}{x}_2 \ominus 0\overset{2\ominus 0}{x}_1\overset{1\ominus 2}{x}_2 \dot{-} 0\overset{2\ominus 2}{x}_2 \dot{-} \\ &\quad \dot{-} 2\overset{1\ominus 0}{x}_1\overset{2\ominus 2}{x}_2 \dot{-} 2\overset{2\ominus 0}{x}_1\overset{2\ominus 2}{x}_2 \\ &= \overset{1}{x}_1 \ominus 1\overset{2}{x}_1 \ominus 1\overset{0}{x}_2 \ominus 1\overset{1}{x}_1\overset{0}{x}_2 \ominus \overset{1}{x}_1\overset{1}{x}_2 \ominus \overset{2}{x}_1\overset{1}{x}_2 . \end{aligned}$$

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 AND, OR, XOR and 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 an 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 operation 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 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 $f : B^n \rightarrow Y$, 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 $g : B^n \rightarrow Y$ is any completely specified function $f : B^n \rightarrow B$ 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 our the computation in the algorithm.

In an algebraic representation of a function, a *literal* is a variable or its comple-

ment. 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:

$$f(x_1, x_2, x_3, x_4) = (x_1' x_2 + x_3' x_4) \dot{+} (x_1 x_2' + x_3 x_4')$$

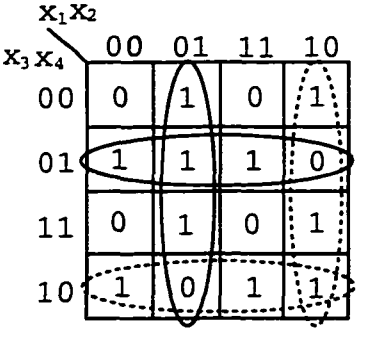


Figure 4.1: Karnaugh map of the function from the example.

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

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

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

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

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:*

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

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

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 $f(x_1, x_2, x_3) = x_1x_2 + x_1'x_2x_3'$ can be written as $f(x_1, x_2, x_3) = p_1 + p_2$, with $p_1 = x_1x_2$ and $p_2 = x_1'x_2x_3'$.

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.

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:

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

Recall, that $f(\underline{x}^0_n)$ and $f(\underline{x}^1_{n+1})$ 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:

$$f(\underline{x}^0_n) = (p_1 + p_2 + \ldots + p_i) \oplus (p_{i+1} + p_{i+2} + \ldots + p_j)$$

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

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:

$$\begin{aligned}
f(x_1, \ldots, x_{n+1}) &= x'_{n+1}\, f(\underline{x}^0_{n+1}) + x_{n+1}\, f(\underline{x}^1_{n+1}) && \{(2.1)\} \\
&= x'_{n+1}\, ((p_1 + \ldots + p_i) \oplus (p_{i+1} + \ldots + p_j)) + \\
&+ x_{n+1}\, ((q_1 + \ldots + q_k) \oplus (q_{k+1} + \ldots + q_l)) && \{\text{substitution}\} \\
&= (x'_{n+1}(p_1 + \ldots + p_i) + x_{n+1}(q_1 + \ldots + q_k)) \oplus \\
&\oplus (x'_{n+1}(p_{i+1} + \ldots + p_j) + x_{n+1}(q_{k+1} + \ldots + q_l)) && \{\text{Property 4.1}\} \\
&= (x'_{n+1}p_1 + \ldots + x'_{n+1}p_i + x_{n+1}q_1 + \ldots + x_{n+1}q_k) \oplus \\
&\oplus (x'_{n+1}p_{i+1} + \ldots + x'_{n+1}p_j + x_{n+1}q_{k+1} + \ldots + x_{n+1}q_l) \\
& && \{\text{Distributivity of } \cdot \text{ over } +\}
\end{aligned}$$

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

□

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:

Given a function $f = (T, D, F)$, find two AND-OR expansions g_1 and g_2 such that $g_1 \oplus g_2$ realizes f whenever f is defined and the total number of product-terms in g_1 and g_2 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_{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 $g1\_best$ and $g2\_best$.
4. Using **DivideEqClasses()**, divide the cubes from ON_list into equivalence classes.
5. Using **SelectPartitioning()**, group the resulting equivalence classes into two sets, T_1 and T_2.
6. Construct $g1\_init = (T_1, F, T_2)$ and $g2\_init = (T_2, F, T_1)$.
7. Starting from $g1\_init$, invoke **SpecifyBoth()** to determine which don't cares in $g1\_init$ and $g2\_init$ 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 $g2\_init$.
10. Repeat steps 5 - 9 for N_{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 */
}
```

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:

$$f(x_1,\ldots,x_4) = x_1'x_2x_3' + x_1'x_2x_4 + x_1'x_3'x_4 + x_2x_3'x_4 + x_1x_2'x_4' + x_1x_2'x_3 + x_1x_3x_4' + x_2'x_3x_4'$$

These product-terms are shown in Figure 4.2.

Suppose that we want to represent this function as $f = g_1 \dot{+} g_2$ 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 $x_1'x_2x_3' \cdot$

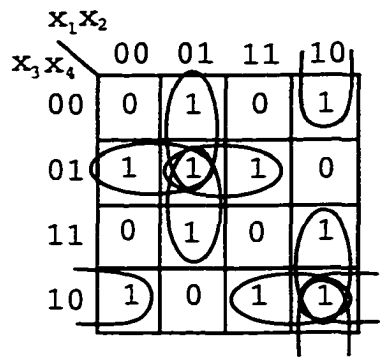


Figure 4.2: Map of the example function.

$x_1'x_3'x_4 = x_1'x_2x_3'x_4$. If p_1 is in g_1 and p_2 is in g_2, when $x_1'x_2x_3'x_4$ is not in $g_1 \dot{+} g_2$, and therefore $g_1 \dot{+} g_2 \neq f$. 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_list* and *OFF_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_list* is less than the number of cubes in *OFF_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 **DivideEqClasses()** is called to divide the *ON_list* cubes into equivalence classes. A sketch of the code is shown in Figure 4.3.

The cubes c_i are divided by **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()**.

$$(c_i, c_j) \in R_1 \text{ iff } (c_1 \cap c_2 \neq \emptyset)$$

where $\emptyset$ denotes an empty set and $\cap$ denotes the *intersection of cubes*, defined by the following table:

	$\cap$	c_1: 0	1	*
	0	0	$\emptyset$	0
c_2	1	$\emptyset$	1	1
	*	0	1	$\emptyset$

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 $\hat{f}$ has only one equivalence class, as for example the function shown in Figure 4.4, then AOXMIN returns the functions *g1_best* and *g2_best* as initialized in step 3 of the algorithm. It

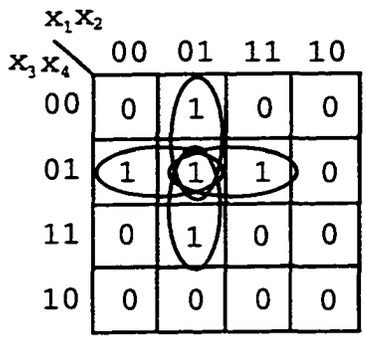


Figure 4.4: A function with all product-terms in the same equivalence class.

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 $g1\_init$ and $g2\_init$

After partitioning the set T into two subsets T_1 and T_2, the functions $g1\_init = (T_1, D_1, F_1)$ and $g2\_init = (T_2, D_2, F_2)$ 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 $g1\_init$ and $g2\_init$ 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 $g1\_init$ and $g2\_init$

After $g1\_init$ and $g2\_init$ 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-

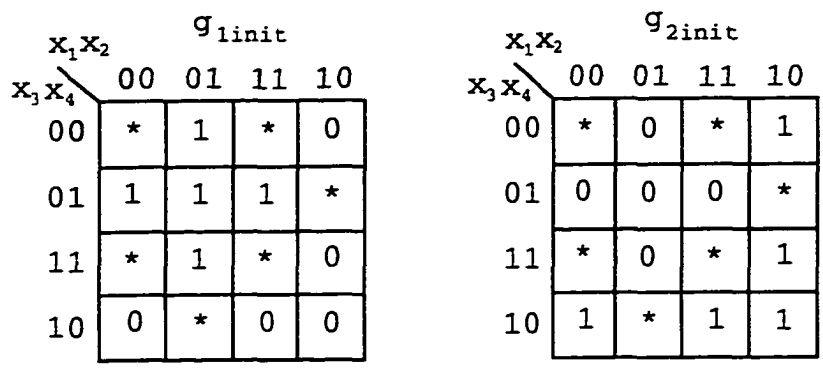


Figure 4.5: Functions *g1_init* and *g2_init* for the function from the example.

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 $g1\_init = (T_1, D_1, F_1)$ and $g2\_init = (T_2, D_2, F_2)$.
output: two AND-OR expansions g_1 and g_2, realizing *g1_init* and *g2_init*, correspondently.

g_1 = 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 = Espresso (T_2, D_2, F_2);
return (g_1, g_2); }

Figure 4.6: Implementation of the subroutine **SpecifyBoth()**.

First, the function $g1\_init = (T_1, D_1, F_1)$ is minimized using Espresso, giving g_1. Then, it is determined which don't cares from *g1_init* were specified to 1 in g_1 by computing the difference $T^* = g_1 - T_1$. These don't cares are specified to 1 in *g2_init* and all other don't cares in *g2_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 *g1_init* and next starting from *g2_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_{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 **DivideEqClasses()**, and thus is $O(N^3)$.

4.4.5 Multiple-output problems

Most digital circuits have multiple outputs. In our current implementation, we consider 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 *g1_best* and *g2_best*. Then we apply Espresso to the resulting $2k$ functions to identify 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 applied 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.

Example function	n	m	Espresso [5]				Alg. [51]	AOXMIN		
			without -Dopo		with -Dopo					
			pr.	t.sec	pr.	t.sec	pr.	pr.	t.sec	N_{iter}
5xp1	7	10	65	0.28	64	1.85	47	42	14.8	10
9sym	9	1	86	0.51	86	1.21	73	73	1.7	1
alu4	14	8	575	29.39	359	204.50	-	447	131.9	1
b12	15	9	43	0.69	29	2.72	-	31	9.1	10
bw	5	28	22	0.62	22	2.01	-	24	25.4	10
clip	9	5	120	1.50	120	8.09	92	95	10.5	10
con1	7	2	9	0.1	8	0.04	-	9	1.0	1
cordic	23	2	914	123.34	155	371.74	-	156	231.8	1
duke2	22	29	86	0.93	86	28.1	-	87	20.2	1
ex1010	10	10	284	42.24	279	202.82	-	725	109.7	1
inc	7	9	30	0.18	28	0.65	-	33	6.5	1
misex1	8	7	12	0.04	12	0.15	-	13	11.5	10
misex2	25	18	28	0.07	28	6.18	-	28	6.0	1
misex3	14	14	690	41.86	189	343.79	-	191	112.0	1
misex3c	14	14	197	13.00	199	108.48	-	197	75.1	10
rd53	5	3	31	0.05	22	0.08	-	19	15.7	20
rd73	7	3	127	0.44	93	1.05	83	83	25.5	10
rd84	8	4	255	1.65	186	3.87	-	192	61.1	10
sao2	10	4	58	0.22	37	1.03	33	38	3.7	1
squar5	5	8	25	0.07	23	0.44	-	22	4.4	1
t481	16	1	481	2.86	481	9.42	364	113	557.2	20
table3	14	14	175	3.81	175	124.50	-	176	79.3	1
table5	17	15	158	2.52	158	179.02	-	158	100.7	1
vg2	25	8	110	0.58	110	71.70	-	102	43.4	10
xor5	5	1	16	0.02	16	0.03	-	10	10.3	20

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_{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

g2_best are:

$$g1\_best = x'_1x_2x_3x'_4 + x'_6x_7x'_8 + x'_5x_6x_8 + x_5x_7x'_8 + x'_5x_6x'_7 + x_1x'_3 + x_1x_4 + x'_2x'_3 + x'_2x_4$$

$$g2\_best = x'_{13}x_{14}x_{15}x'_{16} + x'_{10}x_{11}x'_{12} + x_9x_{11}x'_{12} + x'_9x_{10}x_{12} + x'_9x_{10}x'_{11} + x'_{14}x_{16} + x_{13}x_{16} + x'_{14}x'_{15} + x_{13}x'_{15}$$

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

Table 4.2: AOXMIN results for *t*481.

product-terms	time.sec	N_{iter}
228	47.4	1
134	295.7	10
113	557.2	20
18	1461.5	50

The experimental results indicate that our algorithm works quite well for functions with embedded XOR logic (like 5*xp*1, *clip*, *rd*53, *rd*73, *sao*2, *t*481). On the other hand, for benchmarks without embedded XOR logic, like *misex*1, *misex*2 and *ex*1010, introducing XOR does not bring any advantage. The "hardest" function for our program is *ex*1010, 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.

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 Σ, 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 α 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 α [58]. Such a test $(a_1, \ldots, a_n)$ is said to *detect* the fault α.

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 $\mathcal{T}$ of input vectors is called a *test set* for some set $\mathcal{F}$ of faults if the observation of the corresponding outputs allows the detection of every fault from $\mathcal{F}$ 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:

$$\frac{df(\underline{x})}{dx_i} = f(\underline{x}_i^0) \div f(\underline{x}_i^1)$$

for $i \in N$ [6]. 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

during test generation. We define a generalization of Boolean difference, called *full sensitivity*, not as a quantitative measure of the exact change of $f(\underline{x})$ with respect to a given change of x_i, but as a qualitative measure, reflecting whether $f(\underline{x})$ 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 $\dot{+}$. 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* $f(\underline{x})$ *with respect to* x_i *is defined as*

$$\frac{FSf(\underline{x})}{FSx_i} = mx[f(\underline{x}_i^0), f(\underline{x}_i^1), \ldots, f(\underline{x}_i^{m-1})]$$

where $i \in N$.

If $\frac{FSf(\underline{x})}{FSx_i} \neq 0$, 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 $f(\underline{x})$ changes anytime the value of x_i changes. If $\frac{FSf(\underline{x})}{FSx_i} \neq 0$, then we say that $f(\underline{x})$ is *fully sensitive* to x_i.

If $m = 2$, then

$$\frac{FSf(\underline{x})}{FSx_i} = mx[f(\underline{x}_i^0), f(\underline{x}_i^1)] = f(\underline{x}_i^0) \div f(\underline{x}_i^1),$$

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 $\frac{FSf(\underline{x})}{FSx_i}$, $i \in \{1, 2\}$, may be calculated from the Karnaugh map of the function in accordance with Definition 5.2. Thus, $\frac{FSf(\underline{x})}{FSx_i}$ 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,

$$\frac{FSf(\underline{x})}{FSx_1} = {}^{0}_{x_2} + {}^{1}_{x_2}$$

and

$$\frac{FSf(\underline{x})}{FSx_2} = {}^{2}_{x_1},$$

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 $\frac{FSf(\underline{x})}{FSx_i}$ from its algebraic expansion. To develop such an algorithm, we

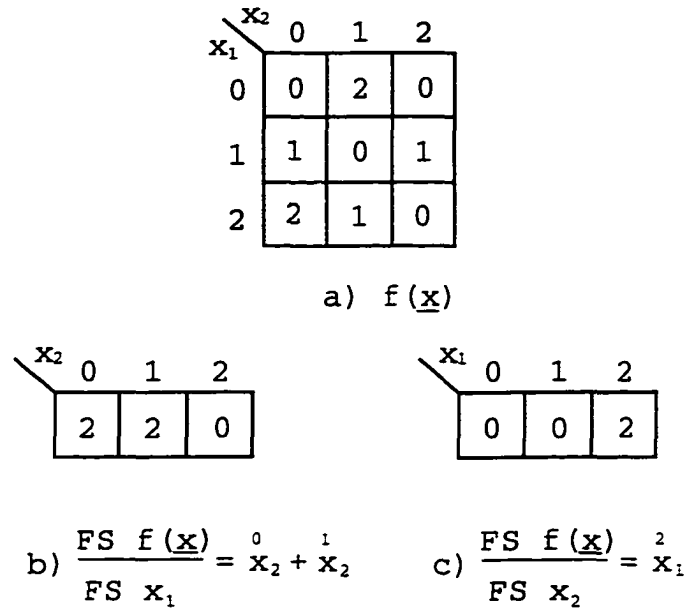


Figure 5.1: Full sensitivities for the example function.

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 Ψ be the set of all permutations over M, i.e.

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

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

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

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

$$= \sum_{(a_0,\ldots,a_{m-1})\in\Psi} \prod_{i\in M} {}^{a_i}x_i$$

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

For $m = 3$:

$$mx(x_0, x_1, x_2) = \overset{0}{x}_0\overset{1}{x}_1\overset{2}{x}_2 + \overset{0}{x}_0\overset{2}{x}_1\overset{1}{x}_2 + \overset{1}{x}_0\overset{0}{x}_1\overset{2}{x}_2 + \overset{1}{x}_0\overset{2}{x}_1\overset{0}{x}_2 + \overset{2}{x}_0\overset{0}{x}_1\overset{1}{x}_2 + \overset{2}{x}_0\overset{1}{x}_1\overset{0}{x}_2 .$$

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:

$$f(\underline{x}) = \overset{0}{x}_i\, f(\underline{x}_i^0) + \overset{1}{x}_i\, f(\underline{x}_i^1) + \ldots + \overset{m-1}{x}_i\, f(\underline{x}_i^{m-1}) \tag{5.1}$$

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

$$f(\underline{x}) = \sum_{k \in M} k \cdot \overset{k}{f}(\underline{x}) \tag{5.2}$$

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

Table 5.1: Truth table for the function from example.

x_1	x_2	$f(x_1, x_2)$	$\overset{0}{x}_1\, f(x_1^0)$	$\overset{1}{x}_1\, f(x_1^1)$	$\overset{2}{x}_1\, f(x_1^2)$	$\overset{0}{f}(x_1, x_2)$	$\overset{1}{f}(x_1, x_2)$	$\overset{2}{f}(x_1, x_2)$
0	0	0	0	0	0	2	0	0
0	1	2	2	0	0	0	0	2
0	2	0	0	0	0	2	0	0
1	0	1	0	1	0	0	2	0
1	1	0	0	0	0	2	0	0
1	2	1	0	1	0	0	2	0
2	0	2	0	0	2	0	0	2
2	1	1	0	0	1	0	2	0
2	2	0	0	0	0	2	0	0

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-

culating $\frac{FSf(\underline{x})}{FSx_i}$. The next property shows that full sensitivities can be calculated as the MIN of m functions.

For notational simplicity, we use $\overset{k}{f}(\underline{x}_i^{\varphi})$ to denote $\sum_{j \in M} \overset{k}{f}(\underline{x}_i^{j})$.

Property 5.5

$$\frac{FSf(\underline{x})}{FSx_i} = \overset{0}{f}(\underline{x}_i^{\varphi}) \cdot \overset{1}{f}(\underline{x}_i^{\varphi}) \cdot \ldots \cdot \overset{m-1}{f}(\underline{x}_i^{\varphi}) = \prod_{k \in M} \overset{k}{f}(\underline{x}_i^{\varphi}).$$

Proof:

$$\begin{aligned}
\tfrac{FSf(\underline{x})}{FSx_i} &= mx(f(\underline{x}_i^0), f(\underline{x}_i^1), \ldots, f(\underline{x}_i^{m-1})) && \{\text{Definition 5.2}\} \\
&= \textstyle\sum_{(a_0,\ldots,a_{m-1}) \in \Psi} \prod_{k \in M} \overset{a_k}{f}(\underline{x}_i^k) && \{\text{Property 5.4}\} \\
&= \textstyle\sum_{(a_0,\ldots,a_{m-1}) \in \Psi} \prod_{k \in M} \overset{k}{f}(\underline{x}_i^{a_k}) && \{\text{Commutativity of MIN}\} \\
&= \textstyle\sum_{(a_0,\ldots,a_{m-1}) \in M^m} \prod_{k \in M} \overset{k}{f}(\underline{x}_i^{a_k}) && \{a \neq b \rightarrow \overset{a}{f}(\underline{x}_i^{j}) \cdot \overset{b}{f}(\underline{x}_i^{j}) = 0\} \\
&= \textstyle\prod_{k \in M} \sum_{j \in M} \overset{k}{f}(\underline{x}_i^{j}) && \{\text{Distrib. of MAX over MIN}\} \\
&= \textstyle\prod_{k \in M} \overset{k}{f}(\underline{x}_i^{\varphi}) && \{\text{Def. of } \overset{k}{f}(\underline{x}_i^{\varphi})\}
\end{aligned}$$

□

There is a simple connection between the functions $\overset{k}{f}(\underline{x}_i^{\varphi})$ and $\overset{k}{f}(\underline{x})$. Expanding $\overset{k}{f}(\underline{x})$ using (5.2) we have:

$$\overset{k}{f}(\underline{x}) = \overset{0}{x}_i \overset{k}{f}(\underline{x}_i^0) + \overset{1}{x}_i \overset{k}{f}(\underline{x}_i^1) + \ldots + \overset{m-1}{x}_i \overset{k}{f}(\underline{x}_i^{m-1}).$$

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

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

1. Express the function $f(\underline{x})$ as (5.2), expanding the functions $\overset{k}{f}(\underline{x})$, $k \in M$, in the MIN-MAX canonical form (Theorem 2.2).

2. In each $\overset{k}{f}(\underline{x})$, replace symbolically all occurrences of the literal $\overset{j}{x}_i$ 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 $\frac{FSf(\underline{x})}{FSx_i}$.

Example 5.6. We compute $\frac{FSf(\underline{x})}{FSx_1}$ for the function from Example 5.3 using the algorithm described above.

1. The function has the following (5.2) form:
$f(\underline{x}) = 0(\overset{0}{x}_1\overset{0}{x}_2 + \overset{0}{x}_1\overset{2}{x}_2 + \overset{2}{x}_1\overset{2}{x}_2 + \overset{1}{x}_1\overset{1}{x}_2) + 1(\overset{1}{x}_1\overset{0}{x}_2 + \overset{1}{x}_1\overset{2}{x}_2 + \overset{2}{x}_1\overset{1}{x}_2) + 2(\overset{0}{x}_1\overset{1}{x}_2 + \overset{2}{x}_1\overset{0}{x}_2)$.
Here, the functions $\overset{0}{f}(\underline{x})$, $\overset{1}{f}(\underline{x})$ and $\overset{2}{f}(\underline{x})$ in the MIN-MAX canonical form are:
$\overset{0}{f}(\underline{x}) = \overset{0}{x}_1\overset{0}{x}_2 + \overset{0}{x}_1\overset{2}{x}_2 + \overset{2}{x}_1\overset{2}{x}_2 + \overset{1}{x}_1\overset{1}{x}_2$
$\overset{1}{f}(\underline{x}) = \overset{1}{x}_1\overset{0}{x}_2 + \overset{1}{x}_1\overset{2}{x}_2 + \overset{2}{x}_1\overset{1}{x}_2$
$\overset{2}{f}(\underline{x}) = \overset{0}{x}_1\overset{1}{x}_2 + \overset{2}{x}_1\overset{0}{x}_2$

2. Replacing all occurrences of literals $\overset{0}{x}_1$, $\overset{1}{x}_1$ and $\overset{2}{x}_1$ by $(m-1)=2$ we have
$\overset{0}{f}(\underline{x}_1^{\phi}) = 2\,\overset{0}{x}_2 + 2\,\overset{2}{x}_2 + 2\,\overset{2}{x}_2 + 2\,\overset{1}{x}_2 = 2$
$\overset{1}{f}(\underline{x}_1^{\phi}) = 2\,\overset{0}{x}_2 + 2\,\overset{2}{x}_2 + 2\,\overset{1}{x}_2 = 2$
$\overset{2}{f}(\underline{x}_1^{\phi}) = 2\,\overset{1}{x}_2 + 2\,\overset{0}{x}_2 = \overset{1}{x}_2 + \overset{0}{x}_2$

3. Applying Property 5.5 we get
$\frac{FSf(\underline{x})}{FSx_1} = 2 \cdot 2 \cdot (\overset{0}{x}_2 + \overset{1}{x}_2) = \overset{0}{x}_2 + \overset{1}{x}_2$.

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 $\frac{FSf(x)}{FSx_i}$ 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 $\overset{k}{x}_i \cdot \frac{FSf(x)}{FSx_i}$ represents a set of tests for all the faults x_i stuck-at-$\overline{k}$, where $\overline{k}$ denotes any value but k, since $\overset{k}{x}_i$ applies the logic value k, different from $\overline{k}$, on the faulty input and the factor $\frac{FSf(x)}{FSx_i}$ 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 $\overset{j}{x}_i \cdot \frac{FSf(x)}{FSx_i}$ 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 $z = f(x_1, x_2) = 1(\overset{1}{x}_1 + \overset{2}{x}_1)\,\overset{1}{x}_2$. In this case $\frac{FSz}{FSx_1} = 0$ and $\frac{FSz}{FSx_2} = 0$ so that tests for single stuck-at faults on the primary inputs cannot be generated using full

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.

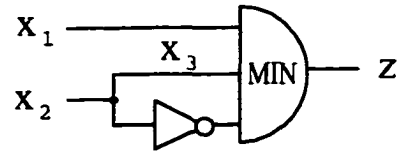


Figure 5.2: Example circuit.

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.

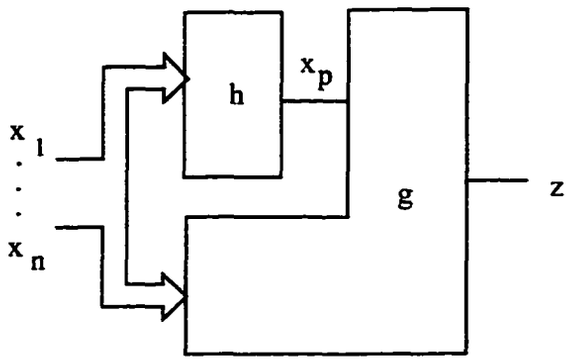


Figure 5.3: A multiple-valued logic circuit.

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.

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

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.

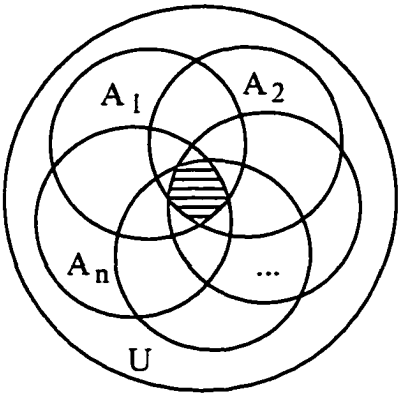


Figure 5.4: Set U with subsets $A_1, A_2, \ldots, A_n$.

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

$$|A_1 \cap A_2 \cap \ldots \cap A_n| \geq |U| - n \cdot (|U| - a), \tag{5.3}$$

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.

$$N_{m,n} = m^{(m^n)}.$$

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:*

$$N_{FS\_to\_x_i} = N_{m,n} - (m^m - m!)^{(m^{n-1})}.$$

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 $j \in \{0, 1, \ldots, m^{n-1} - 1\}$, $mx(f_{mj}, f_{mj+1}, \ldots, f_{mj+m-1}) = 0$. The mutual exclusion operation equals zero when its m arguments are not pairwise distinct. So, the number of m-tuples, satisfying $mx(f_{mj}, f_{mj+1}, \ldots, f_{mj+m-1}) = 0$ 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.

Since the condition $max(f_{mj}, f_{mj+1}, \ldots, f_{mj+m-1}) = 0$ 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

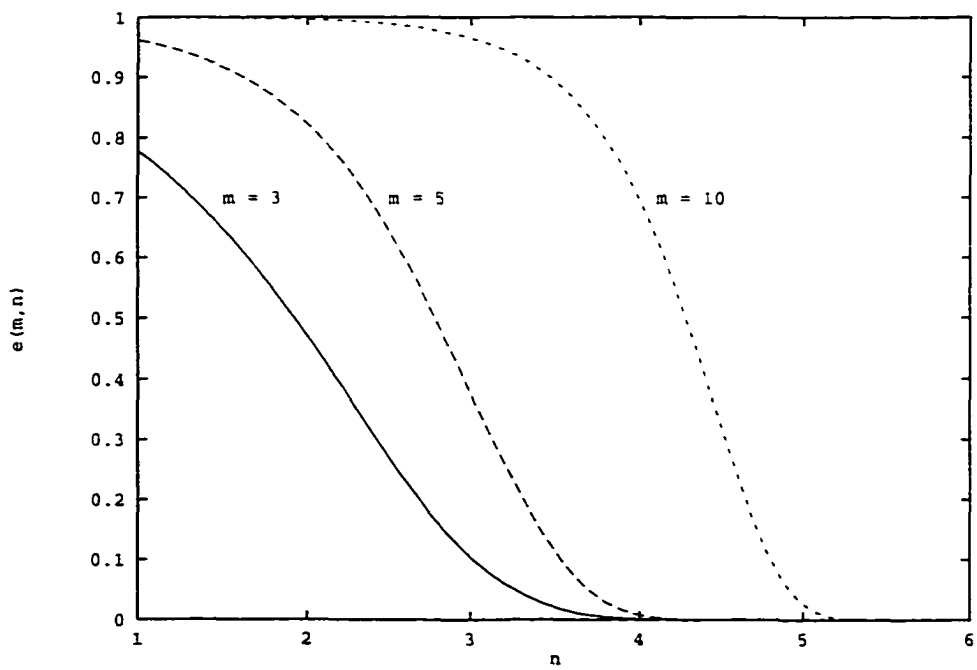


Figure 5.5: Plots for $e(m, n)$ as a function of n for fixed $m = 3, 5$ and 10.

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:

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

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:

$$N_{FS\_to\_all} \geq N_{m,n} - n \cdot (N_{m,n} - N_{FS\_to\_x_i}). \tag{5.4}$$

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:

$$\frac{N_{FS\_to\_all}}{N_{m,n}} \geq 1 - n \cdot (1 - \frac{N_{FS\_to\_x_i}}{N_{m,n}}).$$

Since $\frac{N_{FS\_to\_x_i}}{N_{m,n}} = 1 - e(m,n)$ and $e^*(m,n) := 1 - \frac{FS\_to\_all}{N_{m,n}}$. we have

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

□

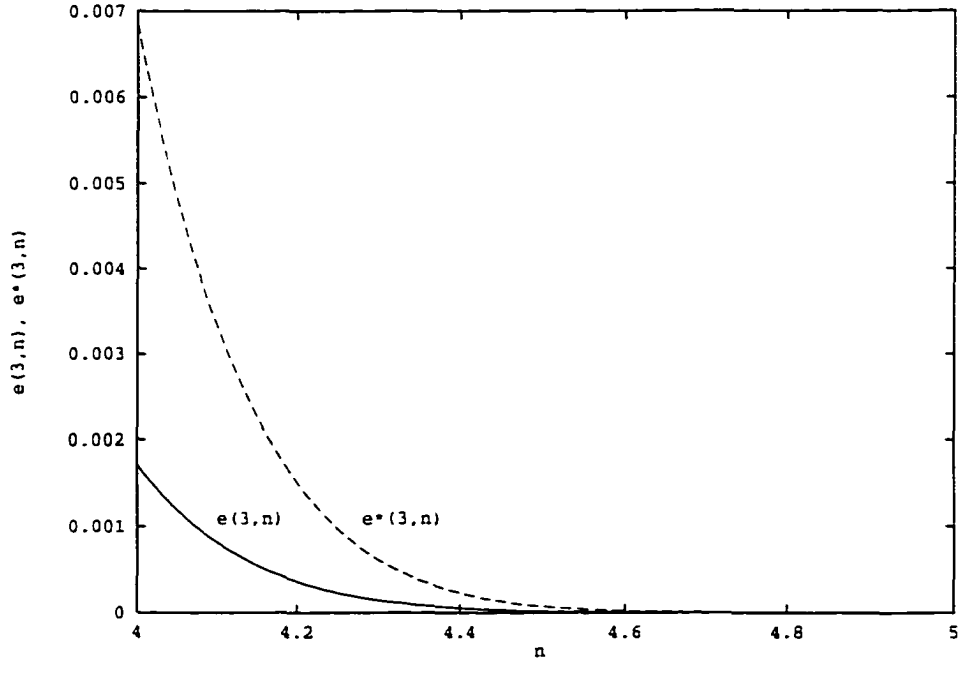


Figure 5.6: Plots for $e(3,n)$ and $e^*(3,n)$ as functions of n for $m = 3$.

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 $1.38 \cdot 10^{-13}$ %. 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 $(3 \leq m \leq 10, \quad n > 5)$ 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-

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

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

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.

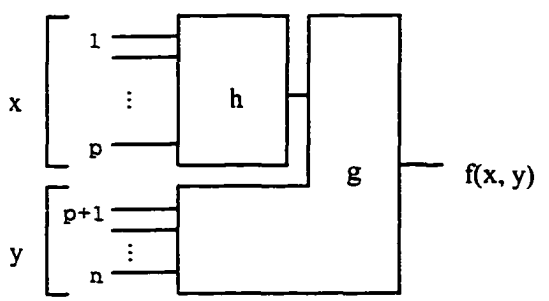


Figure 6.1: Simple disjunctive decomposition

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 $A \in N$. We denote the remaining variables by $y = (x_i)_{i \in \overline{A}}$, where $\overline{A} = N - A$. So, the variables of f can be partitioned and rearranged as (x, y).

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

$$f(x, y) = g(h(x), y) \tag{6.2}$$

for all values of x from M^A, and all values of y from $M^{\overline{A}}$, and suitable functions $h: M^A \rightarrow M$ and $g: M \times M^{\overline{A}} \rightarrow M$. 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.

x_1x_2		00	01	02	10	11	12	20	21	22
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

Figure 6.2: Decomposition chart for an example function in Σ

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]:

$$\begin{array}{ll} \text{multiple:} & f(x, y, z) = g(h(x), k(y), z) \\ \text{iterative:} & f(x, y, z) = g(h(k(x), y), z) \end{array} \tag{6.3}$$

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 Σ 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 $f : M^n \to M$. For this case, the class Σ can be defined as follows.

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

Von Stengel has proved that Σ is closed under composition and decomposition, i.e. if g and h in $f(x,y) = g(h(x),y)$ belong to Σ, then so does f, and vice versa. The class Σ 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 Σ 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 $A \subseteq N$, we say that A is a bound set for f.

A function f that has only the trivial bound sets N and $\{i\}$ for $i \in N$ is called non-decomposable or *prime*, as, for example, always when $n \leq 2$. If $A \subset N$ 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 N 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

$$f(x_1, \ldots, x_6) = g(h(a(x_1, x_2), x_3, x_4), x_5, x_6). \tag{6.4}$$

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.

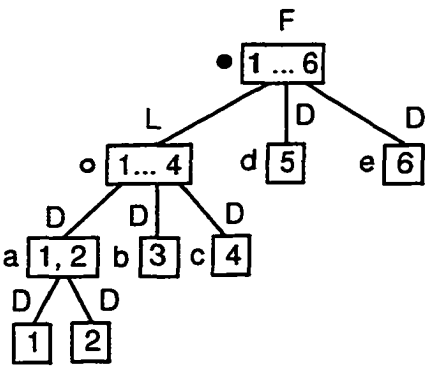


Figure 6.3: Example of a composition tree

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 $f(x, y, z, w) = g(a(x) \bullet b(y) \bullet c(z), w)$ for $x \in M^C$, $y \in M^D$, $z \in M^E$ and remaining variables w where $\bullet$ is an associative function in Σ, 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 $a \bullet b \bullet c$ 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 $\bullet$ (with m^2 values) and to specify a *linear order* among its children showing how to take the product with $\bullet$, like $a \bullet b \bullet c$ since the order $b \bullet a \bullet c$ is not allowed if $\bullet$ is not commutative. If $\bullet$ is commutative, then the order of taking the product is irrelevant. For $m > 2$, Σ may have non-commutative associative operations $\bullet$ like

$$a \bullet b = \begin{cases} a & \text{if } b = 0 \\ b & \text{if } b > 0. \end{cases}$$

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

$$\begin{aligned} h(a(x_1,x_2),x_3,x_4) &= \hat{h}(a(x_1,x_2),b(x_3),c(x_4)) \\ &= a(x_1,x_2) \bullet b(x_3) \bullet c(x_4) \end{aligned} \tag{6.5}$$

with an associative binary operation $\bullet$. The unary functions b and c are bijections 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 $*$ and $\bullet$ are called *isomorphic* if there is a bijection $\varphi: M \to M$ such that $\varphi(a * b) = \varphi(a) \bullet \varphi(b)$.

A *maximal* bound set is an inclusion-maximal bound set not equal to N. Then either

$$\text{all maximal bound sets are pairwise disjoint.} \tag{6.6}$$

or

$$\text{two maximal bound sets } A, B \text{ overlap.} \tag{6.7}$$

In (6.7), $A \cup B = N$ 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*

$$f(y_1, \ldots, y_k) = g(h_1(y_1), \ldots, h_k(y_k)) \tag{6.8}$$

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

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

$$\textbf{TrivialTree}(f, N)$$

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

$$T_1 = T(h), \qquad T_2 = T(g) \tag{6.9}$$

and return

$$T = \textbf{Append}(T_1, A, T_2, \{i\}, \overline{A}).$$

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 $A \cap B$ 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 $A \cap B$ 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

$$\begin{aligned} g(h, y) &= G(h, c(y)), \qquad && h \in M,\ y \in M^{\overline{A}} \\ h(u, v) &= H(a(u), b(v)), \qquad && u \in M^C,\ v \in M^D \end{aligned} \tag{6.10}$$

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

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 $\langle ... \rangle$. The procedure terminates at each **return** statement with the indicated output.

CompositionTree(f, N, S, p)

input: $f \in \Sigma$, $S \subseteq N$, integer p

assumption: $\langle A \subseteq S$ and $|A| \leq p$ for any bound set A of f, $A \neq N \rangle$

output: the composition tree $T = T(f)$

initial call: **CompositionTree**$(f, N, N, n-1)$

1. **if** $|N| \leq 2$ **then**
   return $T :=$ **TrivialTree**(f, N);
2. **while** $p > 1$
   forall $A \subseteq S$ with $|A| = p$
   if IsBoundSet(A, f, N) **then** goto 3;
   end for;
   $p := p - 1$;
   end while;
   $\langle f$ is prime$\rangle$
   return $T :=$ **TrivialTree**(f, N);
3. $\langle A$ is a bound set for f, $|A| = p \rangle$
   let $f(x, y) = g(h(x), y)$ as in (6.2), $i \in A$
   $T_1 :=$ **CompositionTree**$(h, A, A, p-1)$;
   $T_2 :=$ **CompositionTree**$(g, \{i\} \cup (N - A), S - A, \min\{p, |S - A|\})$;
4. **if** $N \neq S$ **or** $p \leq |N| - p$ **or** T_1 or T_2 has a "disjoint" root with > 2 children
   then
   $T :=$ **Append**$(T_1, A, T_2, \{i\}, N - A)$
   else
   $T :=$ **PossiblyMerge**$(T_1, A, T_2, \{i\}, N - A)$;
   return T.

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

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 Σ 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 Σ, which does not include all functions with disjunctive decompositions. An open problem remains how to generalize class Σ so that it includes all disjunctively decomposable functions.

Second, the functions included in class Σ 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 $f: \{0,1\}^n \rightarrow \{0,1\}$, $h: \{0,1\}^p \rightarrow \{0,1,\ldots,m-1\}$ and $g: \{0,1,\ldots,m-1\} \times \{0,1\}^{n-p} \rightarrow \{0,1\}$. In such a decomposition the m-valued function h of 2-valued variables can be coded by $k = \lceil \log_2 m \rceil$ Boolean functions $h_1, h_2, \ldots, h_k$, giving a decomposition of the form

$$f(x, y) = g(h_1(x), h_2(x), \ldots, h_k(x), y) \tag{6.11}$$

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

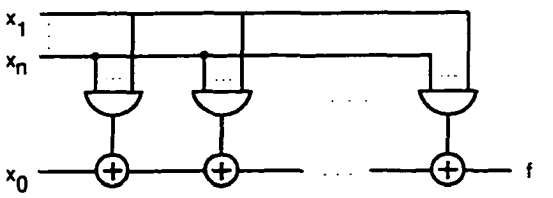


Figure 7.1: Reed-Muller circuit.

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, $i \in \{1, \ldots, m^n - 1\}$. The input x_0 has the value of the constant c_0 during normal operation and a value different from c_0 during testing.

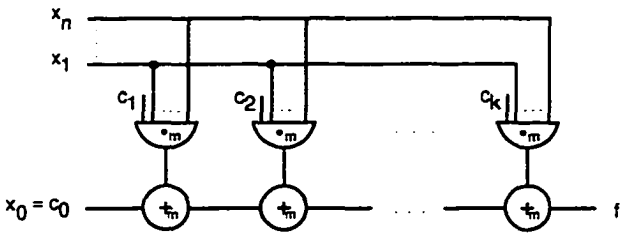


Figure 7.2: Circuit realizing modulo m SOP form.

For example, the 3-variable 3-valued function

$$f(x_1, x_2, x_3) = 1x_1^2x_2 \oplus 2x_1^2x_2^2 \oplus 1x_1x_2x_3 \oplus 2x_1x_2x_3^2$$

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

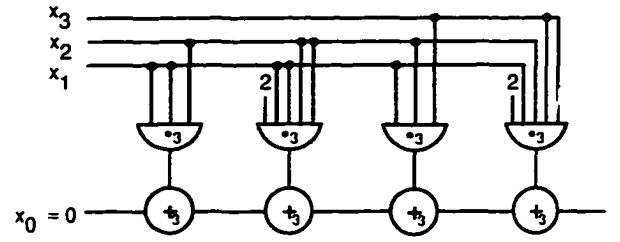


Figure 7.3: Circuit implementing the function from the example.

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 $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$ detects all internal single stuck-at faults. $\mathcal{T}_1$ is defined by the table below

x_0	x_1	x_2	...	x_n
0	0	0	...	0
0	1	1	...	1
1	0	0	...	0
1	1	1	...	1

It detects all single faults on the inputs of XOR gates and all stuck-at-0 faults on the inputs of AND gates. $\mathcal{T}_2$ is defined by $\mathcal{T}_2 := \{\tau_{21}, \tau_{22}, \ldots, \tau_{2n}\}$ with the test τ_{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

inputs of AND gates. So, the $n+4$ tests in the test set $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$ 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:

$$\text{If } a \neq 0 \text{ and } b \neq c \text{ then } ab \neq ac$$

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, than 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 $(11 \ldots 1)$.

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.

x_0	x_1	x_2	...	x_n
0	0	0	...	0
0	1	1	...	1
1	0	0	...	0
0	$m-1$	$m-1$	...	$m-1$

Let us denote by i_k and j_k the inputs of the kth addition mod m gate in the cascade, as shown on Figure 7.4.

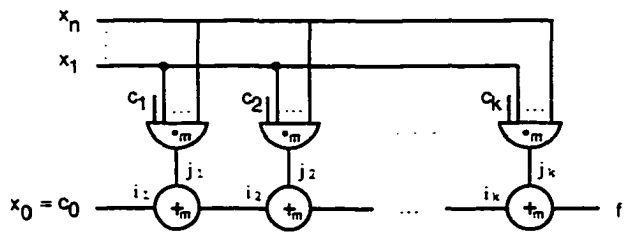


Figure 7.4: Circuit realizing modulo m SOP form.

1. The first test of $\mathcal{T}$ results in applying $(0, 0)$ to each pair (i_k, j_k), detecting all stuck-at-$\overline{0}$ faults on i_k and j_k. By stuck-at-$\overline{a}$ faults we mean all stuck-at-b faults for any $b \in M - \{a\}$.

2. The second test of $\mathcal{T}$ results in applying $(*, c_k)$ to each pair (i_k, j_k), where c_k is the constant (non-zero) which is fed into the kth multiplication mod m gate and $*$ denotes any value from M. It detects all j_k stuck-at-0 faults.

   This test also detects all stuck-at-$\bar{1}$ faults on the inputs of the multiplication mod m gates.

3. The third test of $\mathcal{T}$ 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 $\mathcal{T}$ 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

$$f(a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n) \neq f(a_1, \ldots, a_{i-1}, b_i, a_{i+1}, \ldots, a_n) \qquad (7.1)$$

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 $\mathcal{T} = \{\tau_1, \tau_2\}$ 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, τ_{i1} and τ_{i2} are defined in the following way:

τ_{i1} specifies $x_i = 0$, all other inputs of G_i to 1, and all other primary inputs to 0.

τ_{i2} specifies $x_i = 1$, all other inputs of G_i to 1, and all other primary inputs to 0.

The test τ_{i1} detects x_i stuck-at-1, and the test τ_{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 $X := \{x_j \mid x_j$ is a primary input on which all selected gates

depend};

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 τ_{i1} and τ_{i2} in the following way:

   τ_{i1} specifies $x_i = a_i$, primary inputs in $X - \{x_i\}$ in correspondence with the assignment A, and $x_j = 0$ for all $x_j \notin X$.

   τ_{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 $i \in N$.

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:

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

□

For example, for $m = 3$, $n = 3$ and a variable x_1 we have:

$k = 1$: $\Theta_{1,3} = 2^1 \binom{2}{0} = 2$, product-terms: x_1, x_1^2

$k = 2$: $\Theta_{2,3} = 2^2 \binom{2}{1} = 8$, product-terms: $x_1x_2, x_1x_2^2, x_1^2x_2, x_1^2x_2^2$, $x_1x_3, x_1x_3^2, x_1^2x_3, x_1^2x_3^2$

$k = 3$: $\Theta_{3,3} = 2^3 \binom{2}{2} = 8$, product-terms: $x_1x_2x_3, x_1x_2x_3^2, x_1x_2^2x_3$, $x_1x_2^2x_3^2, x_1^2x_2x_3, x_1^2x_2x_3^2, x_1^2x_2^2x_3, x_1^2x_2^2x_3^2$

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:*

$$\Phi_{k,n} = \frac{1}{m^{\left(\sum_{i=1}^{k} \Theta_{i,n}\right)}}$$

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

$$\frac{m^{(m^n - \sum_{i=1}^{k} \Theta_{i,n})}}{m^{m^n}} = \frac{1}{m^{\left(\sum_{i=1}^{k} \Theta_{i,n}\right)}}$$

□

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 $\frac{1}{m^{\left(\sum_{i=1}^{k} \Theta_{i,n}\right)}}$ 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.

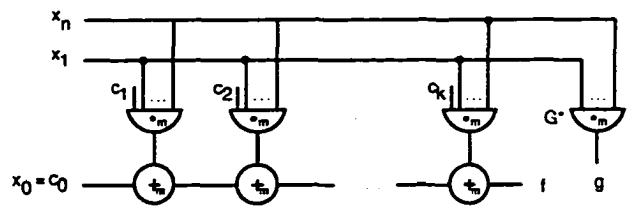


Figure 7.5: Circuit with an extra multiplication mod m gate G^*.

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 $\mathcal{T}$ from Theorem 7.1:

x_0	x_1	x_2	...	x_n
0	0	0	...	0
0	1	1	...	1
1	0	0	...	0
0	$m-1$	$m-1$	...	$m-1$

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 $\mathcal{T}$ 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 depletition-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 depletition-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.

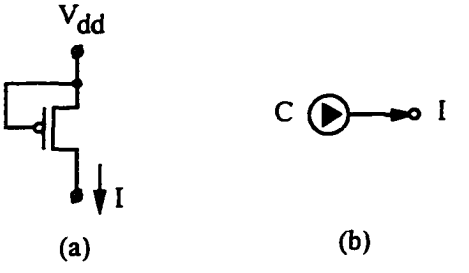


Figure A.1: Current source: (a) circuit configuration, (b) symbol.

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 II, 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 $IO_1, IO_2, \ldots, IO_n$, produced at drains, are determined by input current II and the ratio between output and input W/L values. The ratio is usually chosen so that:

$$IO_i = \begin{cases} II & \text{if } II > 0 \\ 0 & \text{if } II \leq 0. \end{cases}$$

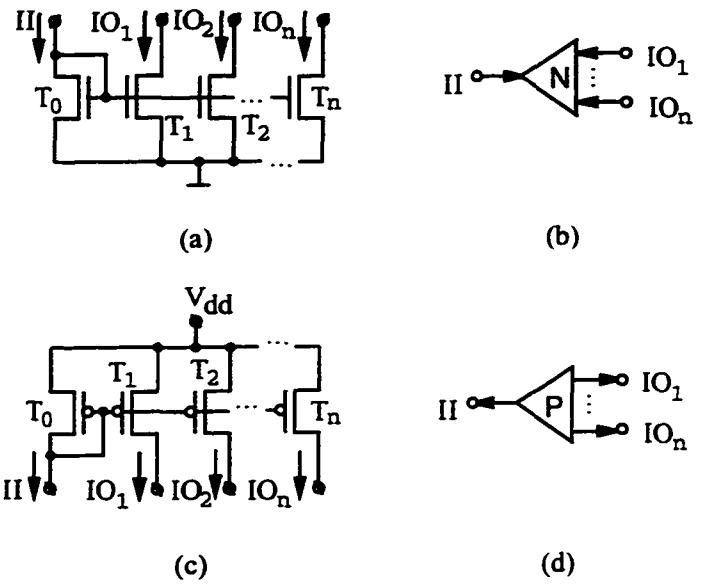


Figure A.2: N-type and P-type current mirrors: (a),(c) circuit configurations, (b),(d) symbols.

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.

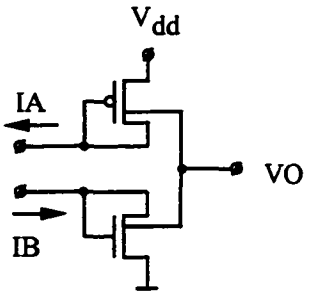


Figure A.3: Classical current comparator.

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 $IB'' = T(IB, IA)$.

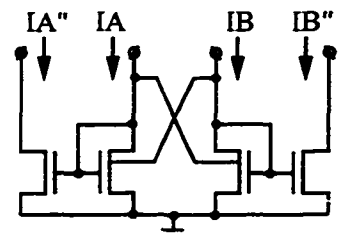


Figure A.4: Threshold current comparator of Onneweer and Kerkhoff.

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

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 VTO=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 VTO=-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 all. [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 15μA 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 max(IA,IB) is obtained. The minimum value MIN(IA,IB) is generated by subtracting the maximum value MAX(IA,IB) from the arithmetic sum of IA and IB. Current mirrors are included to distribute the input currents. The total number of transistors in this circuit is 16.

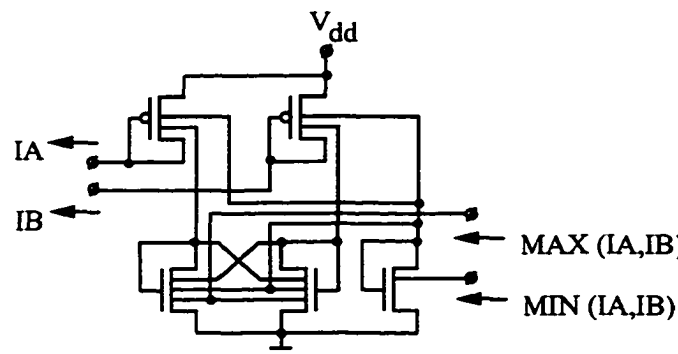


Figure A.5: Minimum-maximum circuit of Onneweer and Kerkhoff.

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 $\oslash$, called *bounded difference* operation be defined as:

$$x \ominus y := \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{if } x < y. \end{cases}$$

where "-" is the regular arithmetic subtraction.

Then

$$MAX(x, y) = x + (y \ominus x) = y + (x \ominus y)$$

$$MIN(x, y) = x - (x \ominus y) = y - (y \ominus x).$$

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

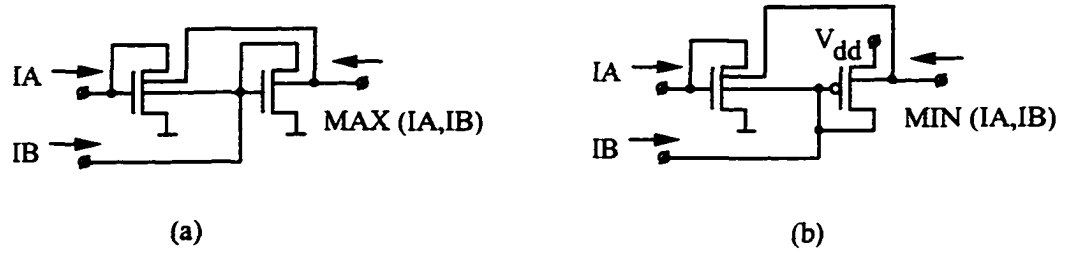


Figure A.6: (a) Minimum and (b) Maximum circuits of Zhijian and Hong.

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].

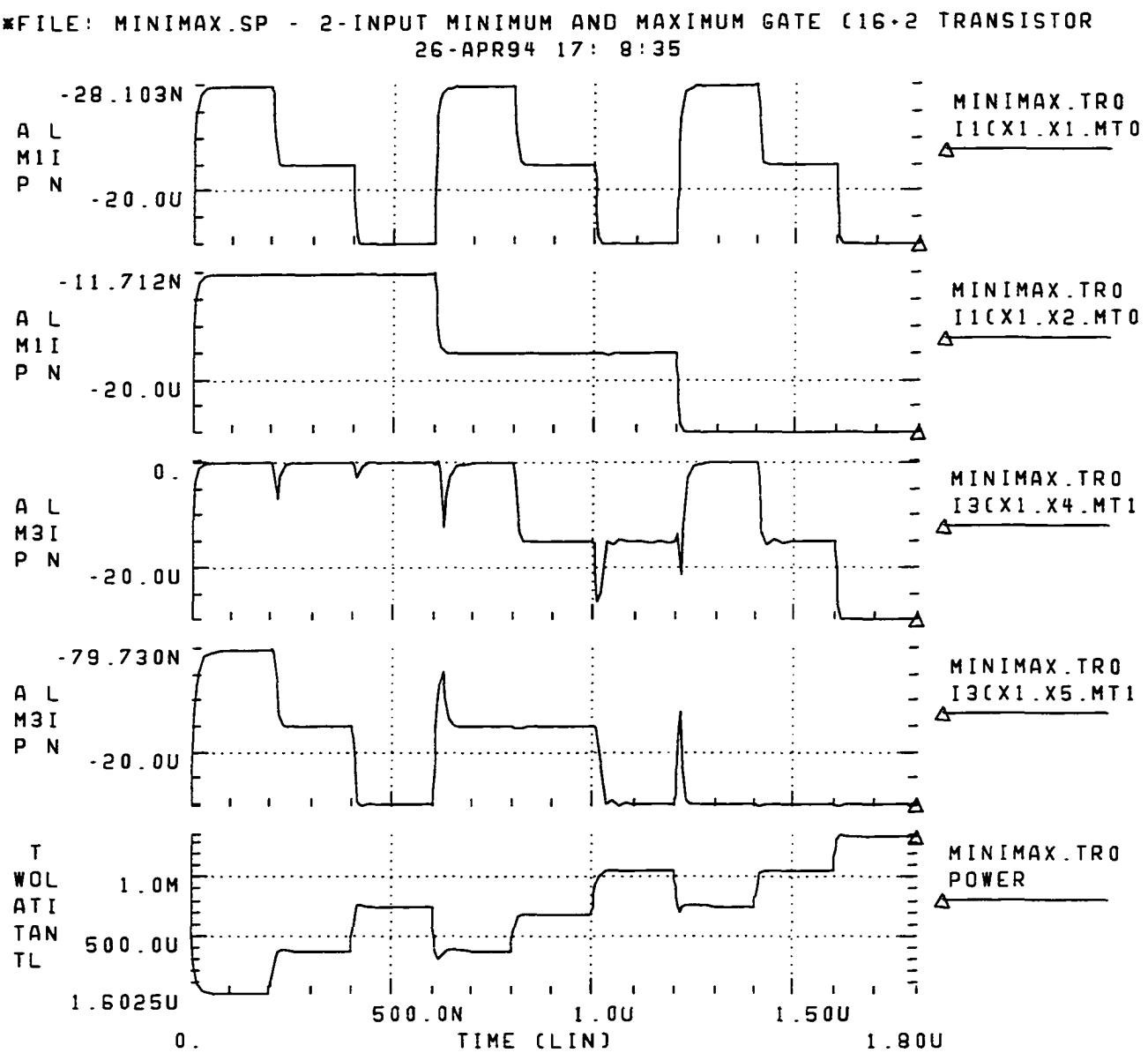


Figure A.7: The simulation results for the MIN/MAX circuit of Onneweer and Kerkhoff: (a) IA, (b) IB, (c) MIN(IA,IB), (d) MAX(IA,IB), (e) power dissipation.

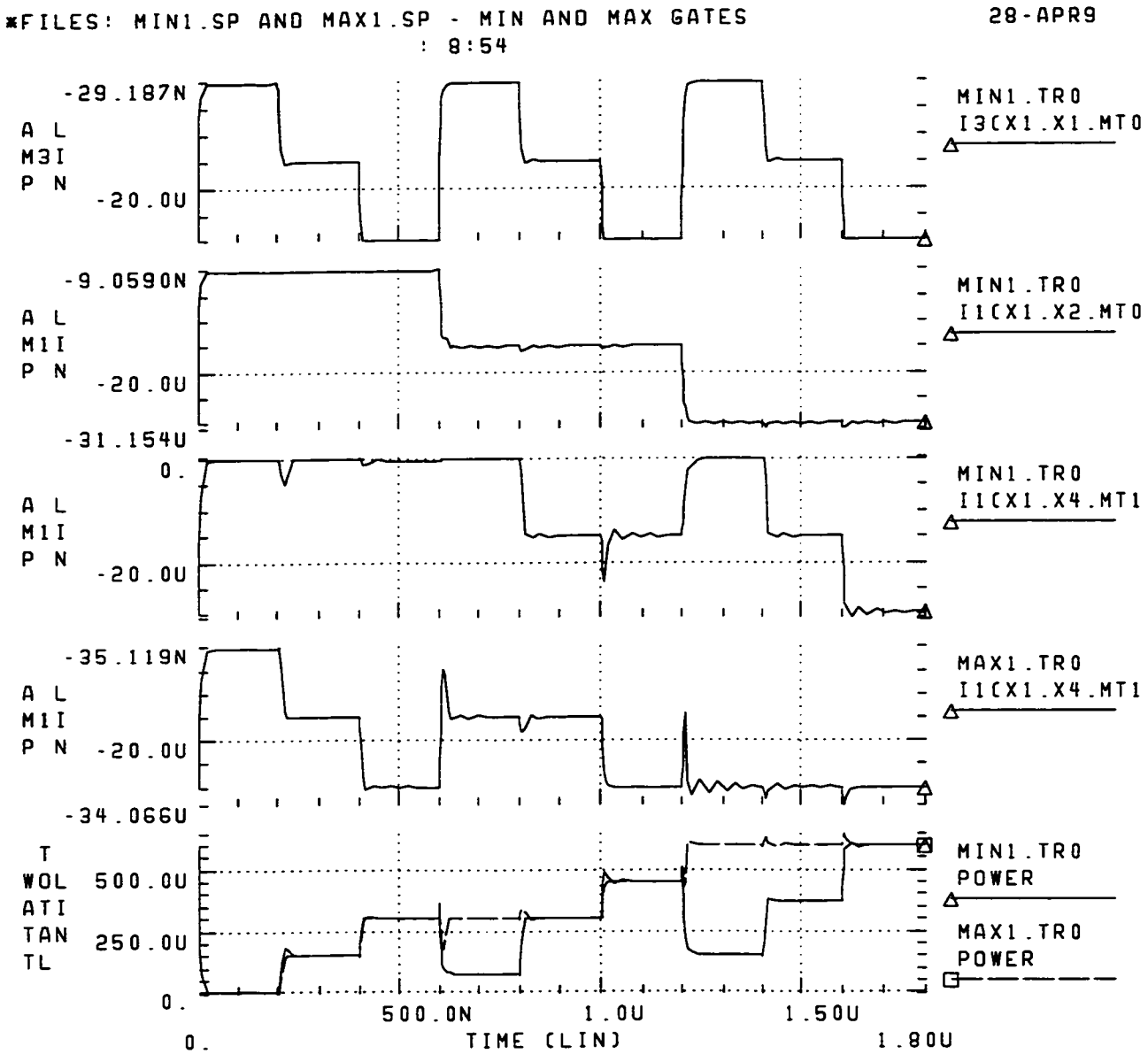


Figure A.8: The simulation results for the MIN and MAX circuits of Zhijian and Hong: (a) IA, (b) IB, (c) MIN(IA,IB), (d) MAX(IA,IB), (e) power dissipation.

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 2 Addition modulo 4 compared to absolute difference.

$\oplus$	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

AD	0	1	2	3
0	0	1	2	3
1	1	0	**1**	2
2	2	**1**	0	1
3	3	2	1	0

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

$$|A - B| = ramp(A - B) + ramp(B - A)$$

where $ramp(x - y)$ is the operation defined as

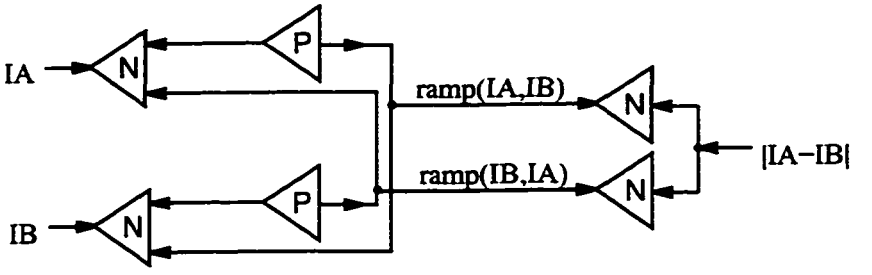
$$ramp(x - y) := \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{if } x < y. \end{cases}$$

where "-" is the regular arithmetic subtraction.

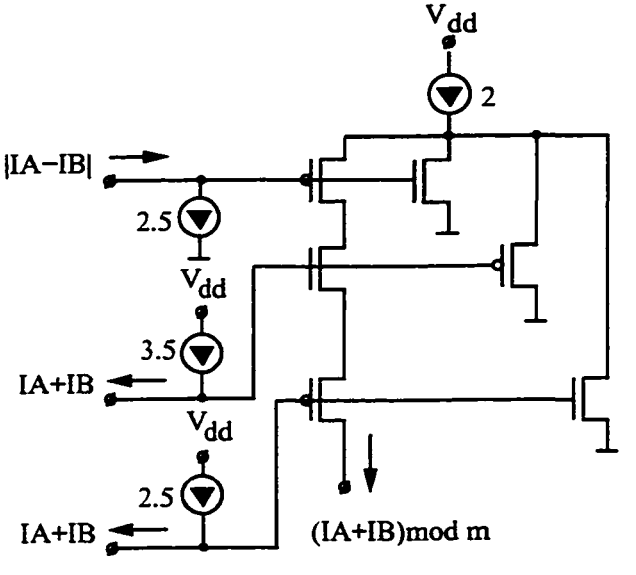
Note the similarity between this operation and the operation $\ominus$ defined above. Figure A.9a shows the realization of the absolute difference circuit.

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.



(a)



(b)

Figure A.9: (a) Absolute difference circuit (b) Correction circuit.

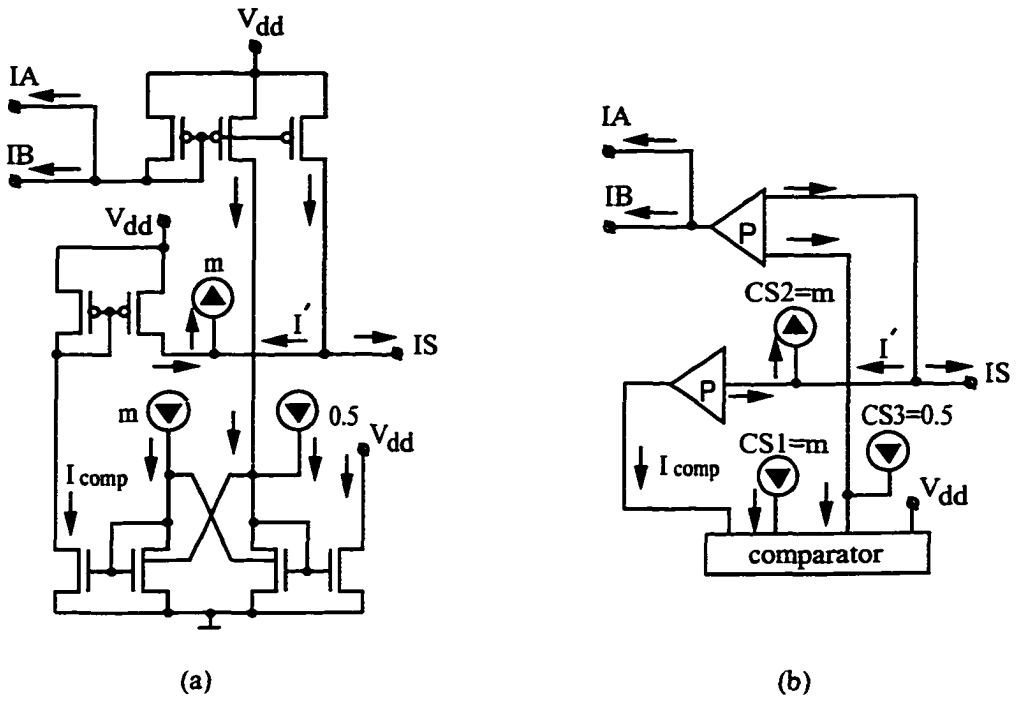


Figure A.10: Addition modulo m circuit (a) circuit configurations. (b) symbolic scheme.

The circuit is based on the threshold current comparator (Figure A.4). The arithmetic sum of two input currents IA and IB is compared with the current which has value m (current units), generated by current source CS1. The output of the current comparator is the current $I_{comp} = T(m, IA + IB)$, i.e.:

$$I_{comp} = T(m, A + B) = \begin{cases} 0 & \text{if } m < IA + IB \\ m & \text{if } m > IA + IB \end{cases}$$

If the sum IA + IB + 0.5 is less than m (current units), then the output current of the current comparator I_{comp} is m (current units). This current "compensates" the m current units, extracted by the current source CS2, and the current I' becomes equal to zero. Thus, the output current of the circuit for this case is IS = IA + IB - 0 = IA + IB.

If the sum IA + IB + 0.5 is greater than m (current units), then the output current of the current comparator I_{comp} is 0. In this case I' becomes equal to m

current units, and these m current units are extracted from the arithmetic sum of IA and IB. Thus, the output of the circuit is IS = IA + IB - I' = IA + IB - m.

The simulation results for this implementation of the addition modulo m circuit for $m = 3$ are shown on Figure A.11 16. 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.

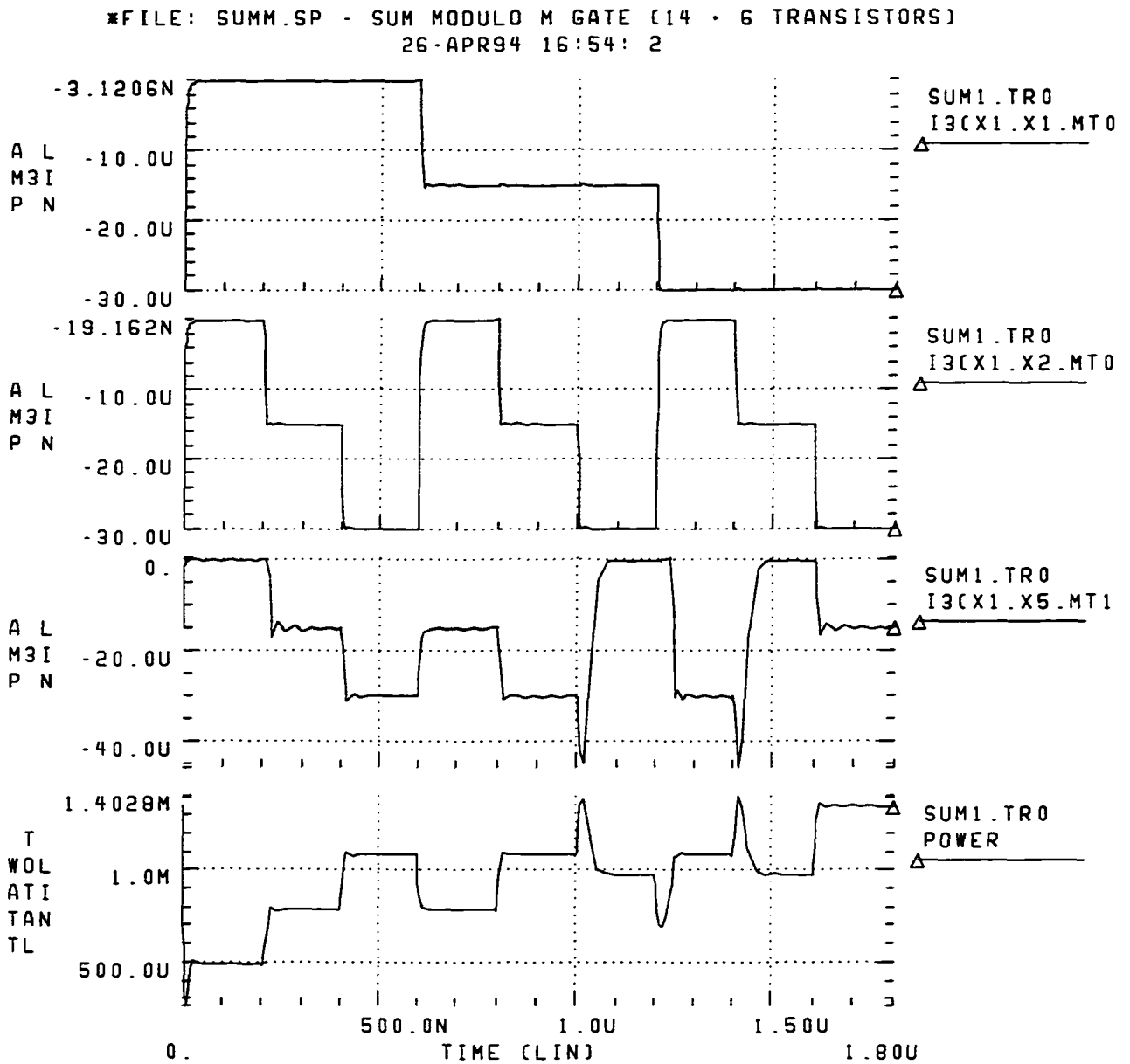


Figure A.11: The simulation results for the addition modulo m circuit: (a) IA, (b) IB, (c) sum(IA,IB), (d) power dissipation.

Bibliography

[1] R. L. Ashenhurst, The decomposition of switching functions, *Proc. International Symp. Theory of Switching* Part I **29** (1959) 74-116.

[2] G. Birkhoff, S. MacLane, *Brief Survey of Modern Algebra*, 4th ed., New York, Macmillan, 1977.

[3] G. Boole, *An Investigation of the Laws of Thought, on Which are Founded the Mathematical Theories of Logic and Probability*, Reprinted by New York: Dover Pub.. 1954.

[4] R. K. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, Multilevel logic synthesis. *Proc. IEEE* **78** No. 2 (1990), 264-300.

[5] R. K. Brayton, G. Hachtel, C. McMullen, A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publisher. 1984.

[6] M. A. Breuer, A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Sci. Press, Inc., Digital Systems Design Series, 1976.

[7] R. L. Burden, J. D. Faires, *Numerical Analysis*, 3rd edition. Prindle. Weber & Schmidt, Boston. 1985.

[8] S. Chattopadhyay, S. Roy, P. P. Chaudhuri, KGPMIN: An efficient multilevel multioutput AND-OR-XOR minimizer, *IEEE Trans. on CAD of ICs and Systems*, **16** No. 3 (1997), 257-265.

[9] M. Cohn, *Switching Function Canonical Form over Integer Fields*, Ph.D thesis, Harvard University, Cambridge, Mass., Dec. 1960.

[10] K. W. Current, A CMOS quaternary threshold logic full adder circuit with transparent latch, *Proc. 20th International Symp. on MVL* (1990), 168-174.

[11] K. W. Current, Multiple valued logic: Current-mode CMOS circuits, *Proc. 23rd International Symp. on MVL* (1993), 176-181.

[12] K. W. Current, D. A. Freitas, F. A. Edwards, CMOS quaternary threshold logic full adder circuits, *Proc. 15th International Symp. on MVL* (1985), 318-322.

[13] H. A. Curtis, *A New Approach to the Design of Switching Circuits*, Van Nostrand, Princeton, 1962.

[14] M. Davio, J.-P. Deschamps, A. Thayse, *Discrete and Switching Functions*, McGraw-Hill International Book Company, Switzerland, 1978.

[15] D. Debnath, T. Sasao, Minimization of AND-OR-EXOR three-level networks with AND gate sharing, *The Sixth Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI '96)* (1996).

[16] D. Debnath, T. Sasao, Exclusive-OR of Two Sum-of Products Expressions: Simplification and an Upper Bound on the Number of Products, *Proc. 3rd International Workshop on the Applications of the Reed-Muller Expansion in Circuit Design* (1997).

[17] E. V. Dubrova, *Test Generation Using Characteristic Differences* (in Bulgarian), M.Sc. Thesis, HIMEE, Sofia, Bulgaria, June 1991.

[18] E. V. Dubrova, D. B. Gurov, J.C. Muzio, Full sensitivity and test generation for multiple-valued logic circuits, *Proc. of 24th International Symp. MVL* (1994), 284-289.

[19] E. V. Dubrova, D. B. Gurov, J. C. Muzio, The evaluation of full sensitivity for test generation in MVL circuits, *Proc. 25th International Symp. on MVL* (1995), 104-109.

[20] E. V. Dubrova, D. M. Miller, J. C. Muzio, Upper bound on number of products in AND-OR-XOR expansion of logic functions, *IEE Journal of Electronics Letters* **31** (1995), 541-542.

[21] E. V. Dubrova, J. C. Muzio, Testability of generalized multiple-valued Reed-Muller circuits, *Proc. 26th International Symp. on MVL* (1996), 56-61.

[22] E. V. Dubrova, J. C. Muzio, Generalized Reed-Muller canonical form for a multiple-valued algebra, *Multiple-Valued Logic, An International Journal* **1** (1996), 65-84.

[23] E. V. Dubrova, J. C. Muzio, B. von Stengel, Finding composition trees for multiple-valued functions, *Proc. 27th International Symp. on MVL* (1997), 19-26.

[24] E. V. Dubrova, D. M. Miller, J. C. Muzio, AOXMIN: A three-level heuristic AND-OR-XOR minimizer for Boolean functions, *Proc. 3rd International Workshop on the Applications of the Reed-Muller Expansion in Circuit Design* (1997).

[25] D. A. Freitas, K. W. Current, A quaternary logic encoder-decoder circuit design using CMOS, *Proc. 13th International Symp. on MVL* (1983), 190-195.

[26] T. A.Guima, M. A.Tapia, Differential calculus for fault detection in multivalued logic networks, *Proc. 17th International Symp. MVL* (1987), 99-108.

[27] B. Harking, C. Moraga, Efficient derivation of Reed-Muller expansions in multiple-valued logic systems, *Proc. 22nd International Symp. on MVL* (1992), 436-441.

[28] P. Ho, M. A. Perkowski, Free Kronecker decision diagrams and their application to atmel 6000 FPGA mapping, *Proc. Euro-DAC* (1994), 8-13.

[29] S. L. Hurst, *The Logical Processing of Digital Signals*, Crane-Russak, New York and Edward Arnold, London, 1978.

[30] S. L. Hurst, D. M. Miller, J. C. Muzio, *Spectral Techniques in Digital Logic*, Academic Press Inc., 1985.

[31] O. Ishizuka, H. Takarabe, Z. Tang, H. Matsumoto, Synthesis of current-mode pass transistor networks, *Proc. 21st International Symp. on MVL* (1991), 139-146.

[32] R. M. Karp, Functional decomposition and switching circuit design, *J. Soc. Indust. Appl. Math.* **11** (1963), 291-335.

[33] S. Kawahito, M. Kameyama, T. Higuchi, H. Yamada, A 32x32-bit multiplier using multiple-valued MOS current-mode circuits, *IEEE Journal of Solid-State Circuits* **23** No. 1 (1988), 124-132.

[34] K. L. Kodandapani, R. V. Setur, Multi-valued algebraic generalization of Reed-Muller Canonical forms, *Proc. 1974 International Symp. on MVL* (1974), 505-544.

[35] H. Lu, S. C. Lee, Fault detection in m-logic circuits using the m-difference, *14th International Symp. MVL* (1984), 62-70.

[36] P. K. Lui, J. C. Muzio, Simplified theory of Boolean functions, *International J. Electronics* **68** (1992), 329-341.

[37] A. A. Malik, D. Harrison, R. K. Brayton, Three-level decomposition with application to PLDs, *IEEE International Conf. on Computer Design* (1991), 628-633.

[38] L. McKenzie, L. Xu, A. Almaini, Graphical representation of generalized Reed-Muller expansions, *Proc. IFIP 10.5 Workshop on Applications of Reed-Muller Expansion in Circuit Design* (1993), 181-187.

[39] D. M. Miller, *Decomposition in Many-Valued Logic Design*, Ph.D. Thesis, University of Manitoba, March 1976.

[40] D. M. Miller, Multiple-valued logic design tools, *Proc. 23rd International Symp. on MVL* (1993), 2-11.

[41] D. M. Miller, J. C. Muzio, Decomposition and the synthesis of many-valued switching circuits, *Proc. 1976 International Symp. on MVL* (1976), 164-168.

[42] R. H. Möhring, Algorithmic aspects of the substitution decomposition in optimization over relations, set systems and Boolean functions, *Annals of Operations Research* **4** (1985), 195-225.

[43] D. E. Muller, Application of Boolean algebra to switching circuit design and to error detection, *IRE Trans. Electron. Comput.* **EC-3** (1954), 6-12.

[44] J. C. Muzio, T. C. Wesselkamper, *Multiple-Valued Switching Theory*, Adam Hilger Ltd Bristol and Boston, 1986.

[45] S. P. Onneweer, H. G. Kerkhoff, Current-mode CMOS high-radix circuits, *Proc. 16th International Symp. on MVL* (1986), 60-68.

[46] W. Paul, Realizing Boolean functions on disjoint sets of variables. *Theoretical Computer Science* **2** (1976), 383-396.

[47] D. K. Pradhan, A multi-valued algebra based on finite fields, *Proc. 1974 International Symp. on MVL*, (1974) 95-112.

[48] S. M. Reddy, Easy testable realizations for logic functions, *IEEE Trans. on Computers* **C-21** No. 11 (1972), 1183-1188.

[49] I. S. Reed, A class of multiple-error-correcting codes and their decoding scheme. *IRE Trans. on Inform. Theory* **4** (1954), 38-42.

[50] J. P. Roth, *Computer Logic, Testing, and Verification*, Rockville. MD: Computer Science Press, 1980.

[51] T. Sasao, A design method for AND-OR-EXOR three-level networks, *Proc. International Workshop on Logic Synthesis*, Lake Tahoe, 1995.

[52] T. Sasao, P. Besslich. On the complexity of mod-2 sum PLA's. *IEEE Trans. on Computers* **39** No. 2 (1990), 262-266.

[53] T. Sasao, J. T. Butler, A design method for look-up table type FPGA by Pseudo-Kronecker expansion, *Proc. of 24th International Symp. MVL* (1994). 342-347.

[54] J. Saul. An algorithm for multi-level minimization of Reed-Muller representation. *Proc. IEEE International Conf. Computer Design* (1991). 634-637.

[55] C. E. Shannon, The synthesis of two-terminal switching circuits. *Bell System Technical J.* **28** (1949), 59-98.

[56] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Trans. A.I.E.E.* **57** (1938), 713-723.

[57] B. von Stengel, *Eine Dekompositionstheorie für mehrstellige Funktionen.* Mathematical Systems in Economics. **123**. Anton Hain, Frankfurt. 1991.

[58] I. G. Tabakow, Using D-algebra to generate tests for m-logic combinational circuits, *International J. Electronics* **75** No. 5 (1993), 897-906.

[59] K. M. Waliuzzaman and Z. G. Vranesic, Decomposition of multiple-valued switching functions, *Computer Journal* **13** (1970), 359-362.

[60] M. Whitney, J. C. Muzio, Decisive differences and partial differences for stuck-at fault detection in MVL circuits, *Proc. 14th International Symp. MVL* (1984), 321-328.

[61] L. F. Wu, M. A. Perkowski, Minimization of permuted Reed-Muller trees for cellular logic programmable gate arrays, *Proc. FPGA-92* (1992), 7/4.1-7/4.4.

[62] T. Yamakawa, CMOS multivalued circuits in hybrid mode, *Proc. 15th International Symp. on MVL* (1985), 144-151.

[63] T. Yamakawa, T. Miki, F. Ueno, The design and fabrication of the current-mode fuzzy logic semi-custom IC in the standard CMOS IC technology, *Proc. 15th International Symp. on MVL* (1985), 76-82.

[64] Z. Zilic, Z. Vranesic, Current-mode CMOS Galois field circuits, *Proc. 23rd International Symp. on MVL* (1993), 245-250.

[65] L. Zhijian, J. Hong, A CMOS current-mode high speed fuzzy logic microprocessor for a real-time expert system, *Proc. 20th International Symp. on MVL* (1990), 394-400.

[66] *The Programmable Logic Data Book*, Xilinx Corporation, Santa Jose, CA, 1994.

[67] *Flash Density Offers Two for One*, Solutions OEM, Intel Corporation, Santa Clara, CA, Fall 1994, p. 1.