Autocorrelation and Decomposition Methods in Combinational Logic Design

by

Randal Wade Tomczuk

B.C.Sc., University of Manitoba, Canada, 1983
M.Sc., University of Manitoba, Canada, 1986

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. D. M. Miller, Supervisor (Department of Computer Science)

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

Dr. H. Müller, Departmental Member (Department of Computer Science)

Dr. N. Dimopoulos, Outside Member (Department of Electrical and Computer Engineering)

Dr. J. T. Butler, External Examiner (Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, CA)

Supervisor: Dr. D. M. Miller

Abstract

This dissertation shows that the autocorrelation of switching functions can be effectively utilized in combinational logic optimization and synthesis. The procedures developed exploit information contained in the autocorrelation of switching functions to perform optimization of Programmable Logic Arrays (PLAs) and to aid in a multi-level logic synthesis approach called two-place decomposition.

A new optimization technique is presented, based on the autocorrelation of switching functions, to find near-optimal variable pairings for decoded PLAs. The results of this approach compare favourably to those of other researchers' techniques. The key advantages of the new approach are its simplicity and its efficiency.

The basic two-place decomposition approach is augmented with various enhancements. These include an improved decomposition merge procedure, the addition of alternate mapping functions for complex disjunctive decompositions, and the incorporation of linearization using the autocorrelation to handle functions that are non-two-place decomposable. A robust implementation of the enhanced method is presented and is used to generate function realizations for comparison with other synthesis methods. The enhanced two-place decomposition method is shown to perform particularly well for functions exhibiting high degrees of symmetry.

The dissertation also presents a new synthesis technique that utilizes a particular representation of a switching function called a *Reduced Ordered Binary Decision Diagram* (ROBDD) and is targeted to two-place decomposition. This new technique allows the two-place decomposition approach to synthesize a much broader range of functions. Although, in comparison to one other synthesis method, the new ap-

proach does not perform as well in most cases. it has considerable promise and several enhancements are proposed for improvement.

This dissertation also shows that there is a strong connection among autocorrelation, two-place decomposition. and good variable orders in an ROBDD. A first attempt to formally analyze the relationship between autocorrelation and two-place decomposition is presented. Relationships are identified between certain autocorrelation coefficients when particular two-place decompositions exist in a function. These relationships are also connected to the heuristics used in the above mentioned PLA optimization technique.

Variable order can have a substantial impact on the size of an ROBDD. This dissertation shows that a good variable order is related to the two-place decompositions that are exhibited in a function. Thus. variable order is also related to the autocorrelation and this relationship can lead to an autocorrelation-based technique for determining good variable orders for ROBDDs.

Examiners:

Dr. D. M. Miller. Supervisor (Department of Computer Science)

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

Dr. H. Müller. Departmental Member (Department of Computer Science)

Dr. N. Dimopoulos. Outside Member (Department of Electrical and Computer Engineering)

Dr. J. T. Butler. External Examiner (Department of Electrical and Computer Engineering. Naval Postgraduate School. Monterey. CA)

Table of Contents

List of Algorithms

List of Tables

List of Figures

Acknowledgments

I am grateful to my supervisor, Dr. D. M. Miller, for his help and guidance throughout the course of this work. I also thank Dr. H. Müller, who provided invaluable assistance during the final preparation of this dissertation. Also I would like to express my appreciation to Drs. J. T. Butler, J. C. Muzio, and N. Dimopoulos for their extensive comments and suggestions, which made for a much improved document.

In addition, I wish to thank the Departments of Computer Science at the University of Victoria and at the University of British Columbia for the use of their facilities.

I gratefully acknowledge the financial support from an NSERC Postgraduate Scholarship and from an ASI Graduate Scholarship.

There is no way I can adequately acknowledge the influence of my parents and family in shaping my life. I thank them for their warmth, their intelligence, their quiet dignity and courage, their remarkable spirit of survival, and their unwavering support.

Last and most significantly, my love and gratitude go to my wife Jody for her confidence and encouragement throughout the writing of this dissertation. I thank her for her continual support, understanding, patience, and prodding, for her seriousness as well as her sense of humour. She is as excited as I am that it is done.

To Jody

Chapter 1

Introduction

Synthesis and optimization of switching circuits is a complex problem and finding exact solutions is infeasible. Hence most approaches to these tasks are heuristic in nature and generate "good" or "acceptable" solutions in reasonable time. In certain cases a good or acceptable solution is one that meets certain constraints that may be placed on the circuit such as a minimum speed or maximum area and it may not be important to achieve the fastest possible or smallest possible circuit.

One drawback of heuristic approaches is that they tend to perform well on some classes of functions and perform poorly, sometimes extremely poorly, on other functions. For example, a synthesis heuristic may work extremely well on symmetric functions but may not work at all well on functions exhibiting no symmetry.

Since it is likely that no one tool will be applicable to all problem areas or even to all cases of the same problem, it is important to have a suite of tools on hand so that the hardware designer may apply the appropriate one to a particular optimization or synthesis problem. If the designer is unable to determine, from knowledge, experience or otherwise, which tool is appropriate, then the tools should perform relatively quickly so that several may be applied and the best results used.

The procedures developed in this dissertation fit into this scheme. They are heuristic based and as such do not necessarily arrive at optimum solutions and may even perform badly at times. However, they are relatively fast and produce good results compared with those reported in the literature.

This dissertation investigates new applications of the *autocorrelation* of switching functions in optimization and synthesis of combinational logic. The procedures developed here use information contained in the autocorrelation of switching functions to perform PLA optimization and to aid in a particular multi-level synthesis approach, namely two-place decomposition. It also investigates the relationships among autocorrelation, two-place decomposition and some characteristics of a particular representation of a switching function called a *Binary Decision Diagram* (BDD).

Section 1.1 provides an overview of the autocorrelation of switching functions and remaining sections briefly describe the problems that are addressed by this dissertation and provide overviews of the results. The chapter concludes with a summary of the major contributions of this work.

1.1 Autocorrelation

Fourier analysis [12, 57] is used extensively in engineering. The premise behind Fourier analysis is that various phenomena may be described as a sum of periodic functions, which typically are sine and cosine functions.

The equivalent of Fourier analysis for switching functions uses transformations based on an orthogonal set of functions; for example, the Rademacher-Walsh [76, 96] functions correspond to the Boolean exclusive-OR functions and are the discrete analogue of the continuous sine and cosine functions used in Fourier analysis.

Consider a switching function $f(X)$ of n variables. Each row in a truth table defining $f(X)$ completely defines the behaviour of the function for a single set of input

values but provides no information about its behaviour anywhere else. The combination of the behaviour specified for all the 2^n possible input assignments provides a complete behavioural definition of the function.

Like the truth table, the *spectrum* of a switching function contains 2^n values, which are called *spectral coefficients*. However, each coefficient contains some information about the global behaviour of the function over all input assignments but almost no information about the local behaviour for a single input assignment. Various properties that are difficult to recognize in the Boolean domain are more easily seen in the spectral domain because of this global information.

A construct closely related to the spectrum of a switching function is its *autocorrelation*. The autocorrelation coefficients provide global information regarding areas of similarity within the function. While the autocorrelation function is widely used in other fields, it has not been as extensively applied in the area of switching function optimization and synthesis.

Rademacher [76] and Walsh [96] are the originators of the spectral transform matrices used in the spectral techniques discussed in this dissertation. These matrices are examples of particular Hadamard [95] matrices.

Spectral techniques were first applied in the areas digital signal processing and transmission of information [33]. Application of spectral techniques in the area of digital logic analysis, design and synthesis was first considered by Coleman [20]. Several books dealing with these topics subsequently have been published [10, 11, 37, 39, 48, 54]. Lechner's work [53, 54] has led to the use of spectral techniques to develop function classification methods, which can be used in synthesis procedures [37, 39, 90] (see Chapters 2 and 6).

More recently, spectral and autocorrelation techniques have been employed in circuit testing [1, 39, 65]. The reader is directed to these references for discussion of spectral testing techniques since these techniques are beyond the scope of this

dissertation.

Chapter 2 provides the formal background for the spectrum and autocorrelation along with the necessary background information and definitions for the other topics discussed in this dissertation. General terms and notation that are used in this dissertation are also provided. The chapter introduces switching functions and their representation, manipulation, and realization in hardware. It then defines the spectra and autocorrelation of switching functions and provides an overview of the autocorrelation based procedures that are developed in this dissertation.

1.2 PLA Optimization

A *Programmable Logic Array* or *PLA* is a regularly structured two-level AND/OR circuit that realizes directly a set of sum-of-products expressions describing a system of switching functions. Efficient minimization methods [13, 23, 36, 82] exist for determining minimal or near-minimal expressions that are targeted to PLA implementation.

Chapter 3 introduces a new approach to solving a problem in PLA optimization. namely, the problem of finding near-optimal variable pairings for decoded PLAs. The pairing selection heuristic uses information contained in the autocorrelation of a system of switching functions to choose the pairings.

The new approach is compared with techniques proposed by Sasao [81, 82, 84], and Chen and Muroga [18]. As seen in Chapter 3, the autocorrelation pairing procedure yields results that are comparable to those of these other approaches. However, there are instances where the results obtained are inferior, again, due to the heuristic nature of the procedures involved. Two variants of the basic pairing procedure are developed to address these situations.

As described in Chapter 3, the key advantages of the new approach are its relative simplicity and its efficiency. For most of the functions considered, the procedure is considerably faster than the other procedures. This allows the user or another logic optimization or synthesis tool to try any or all of the variants of the autocorrelation pairing procedure and use the best results that are obtained.

1.3 Two-place Decomposition

Multi-level logic synthesis is introduced in Chapter 4. It provides a brief review of each of the general methods that are used to produce multi-level realizations for switching functions, putting particular emphasis on *decomposition* techniques.

Decomposition is the re-expression of a function as a composition of simpler sub-functions. A multi-level realization is created by determining a sequence of decompositions, each of which expresses a circuit in terms of continually simpler sub-circuits, until the function is entirely expressed in terms of primitive switching elements, usually logic gates.

Two-Place Decomposition, a restricted form of decomposition, is the subject of Chapters 5 and 6. Chapter 5 presents two-place decomposition and describes several enhancements to the basic approach. The enhanced procedure is implemented as a C [50] program referred to as Decomp, which is used in the experiments described in Chapters 5 through 7. Chapter 5 pays particular attention to symmetric functions and the results of applying Decomp are compared to those found using two other synthesis systems: Oasis [15] and MIS [14]. Decomp is found to produce excellent results for this class of functions.

In Chapter 6 the autocorrelation of switching functions is used to extend the capabilities of the two-place decomposition to functions that do not exhibit symmetries. *Linearization* using the total autocorrelation of the functions is performed to attempt

to convert these functions into functions that do exhibit symmetry. The chapter contains a comparison of the enhanced procedure with MIS.

The effects of linearizing functions prior to other synthesis is analyzed and it is found that although the linearization process tends to reduce the size of two-level realizations of functions, it has little benefit for multi-level realizations. Indeed, in the vast majority of cases, it increase the sizes of multi-level realizations generated by Decomp.

This observation leads to an analysis of the autocorrelation coefficient values corresponding to an exhibited two-place decomposition. Relationships are identified between values of certain of the autocorrelation coefficients when particular two-place decompositions exist in a function. These relationships are novel and this is the first attempt to formally analyze the relation between autocorrelation and two-place decomposition.

An approach to computing the autocorrelation of an incompletely specified function is proposed in Chapter 6, where the autocorrelation is computed as the cross-correlation of two completely specified functions derived from the original incompletely specified function.

Because of the heuristics used, Decomp produces excellent results in some cases and poor results in others. Thus the user is allowed to specify how the procedure is to proceed and decide which configuration is best for a particular function, or he may try several different configurations and use the best results obtained. This is the approach taken by MIS [14], ITEM [46, 47] and other synthesis systems where an interactive approach allows the user's knowledge and experience to help guide the synthesis process.

1.4 ROBDD Techniques

Chapter 7 considers a particular representation for switching functions called the *Reduced Ordered Binary Decision Diagram* or ROBDD [17]. An ROBDD is a rooted directed acyclic graph (DAG) with a structure similar to that of a binary tree, but nodes can have more than one incoming edge so that paths are not disjoint. This chapter discusses ROBDD-based techniques for switching function representation and manipulation and for spectrum and autocorrelation representation, computation, and manipulation.

This chapter presents a new ROBDD-based synthesis technique targeted to two-place decomposition. The procedure enables the synthesis of functions for which a realization cannot be generated by two-place decomposition alone. It also allows the two-place decomposition procedure to handle large problems since a large problem is partitioned into several smaller ones.

The results of this new synthesis algorithm are compared with results generated by MIS [14]. Although the results are not as good as MIS's in most cases, the method has considerable promise and enhancements to the basic procedure are suggested.

Chapter 7 also establishes relationships between two-place symmetries, two-place decompositions, optimal variable ordering in ROBDDs, and the autocorrelation. The results show that the variables involved in a two-place decomposition should be placed together in an ROBDD. The order of preference for decomposition types is the same as that used in the two-place decomposition procedure.

Use of an autocorrelation-based technique to find a good variable ordering [21] is summarized. The best of four variants that are used relates to the theorems of Chapter 6 that establish relationships between certain autocorrelation coefficients when particular two-place decompositions are exhibited by a function. The use of these theorems and the autocorrelation to compute measures of closeness to decomposi-

tions is suggested as a basis for a variable ordering algorithm.

Conversely, given a good variable ordering, one may identify two-place decompositions directly from an ROBDD. This could certainly lead to a more efficient two-place decomposition procedure. This is also briefly discussed.

1.5 Contributions

The major contributions of this dissertation are:

1. PLA optimization:
   - development of a new algorithm, based on the autocorrelation of switching functions, for finding near-optimal variable pairings for decoded PLAs.
2. Two-place decomposition:
   - enhanced procedure for merging decompositions for several outputs of a system of switching functions;
   - addition of alternate mapping functions for a particular type of two-place decomposition called a complex disjunctive decomposition;
   - incorporation of linearization, based on the autocorrelation, into the two-place decomposition procedure for handling functions that are not two-place decomposable;
   - a robust implementation (Decomp) of two-place decomposition incorporating pre-linearization, full and partial linearization as options, and other enhancements to the basic two-place decomposition method;
   - theorems establishing relationships between certain autocorrelation coefficients for particular two-place decompositions exhibited by a function:

- approach for computing the autocorrelation of an incompletely specified function encoded as two completely specified functions.

3. ROBDD-techniques:

   - ROBDD-based synthesis technique targeted to two-place decomposition:
   - identification of relationships between two-place decompositions, variable order in an ROBDD, and autocorrelation.

The dissertation concludes in Chapter 8 with an assessment of these contributions and suggestions for further research.

Chapter 2

Background

This chapter provides the background material required for the topics discussed in this dissertation. It begins in Section 2.1 by describing switching functions and discussing their manipulation and representation. Also provided is a brief description of certain methods of realizing switching functions as electronic circuits is provided.

In Section 2.2 the spectra of switching functions are described. A function's spectrum provides information about the function that is not readily apparent in the function's truth table representation and this information can be used to advantage in the analysis and synthesis of switching functions. The section provides methods of computation and describes some uses for a function's spectrum.

A closely related construct to a function's spectrum is a function's autocorrelation, which provides information regarding areas of similarity within the function. This information too is not easily seen in the function's truth table. Section 2.3 defines the autocorrelation of a system of switching functions and describes some uses for it. Several of the topics relating to a function's spectrum also apply to a function's autocorrelation. In addition the autocorrelation may be computed from the spectrum more efficiently than from the function's truth table.

The autocorrelation is utilized in Chapter 3, to perform a certain optimization for PLAs, and in Chapters 5 through 7, to assist in performing multi-level synthesis.

2.1 Switching Functions

Digital circuits are known as switching circuits because they behave like switches: transistors are either on (conducting) or off (non-conducting). Theses switches are controlled using binary signals which may have one of two logical values: logic-1 or logic-0. These logical values are also called "true" and "false", "on" and "off", or "high" and "low".

Boolean algebra is the basic mathematical structure used to describe, analyze and synthesize binary switching circuits. When applied to the study of switching circuits, Boolean algebra is often called *switching algebra.*

The desired behaviour of a digital circuit is described using *switching functions.* These functions are manipulated using the rules of switching algebra to produce various results. The following sections provide basic information regarding switching functions and how they may be represented and manipulated.

2.1.1 Basic Definitions and Notation

A *binary vector* is an n-tuple $b_n b_{n-1} \ldots b_1$, $b_i \in \{0,1\}, i = 1, 2, \ldots, n$. Each b_i is called a *bit.* Any binary vector may be interpreted in several equivalent ways: as an n-tuple of elements $\in \{0,1\}$, as a set where each 1-bit indicates that the corresponding item exists in the set, or, as the binary representation of an integer value k, where

$$k = \sum_{i=1}^{n} b_i 2^{i-1}.$$

For example, the five-bit binary vector $u = 01101$ represents the value 13.

The *Hamming distance* (or simply distance) between two binary vectors u and v, denoted by $d(u, v)$, is the number of positions in which they differ. The weight of a binary vector v, denoted by $\|v\|$ is the number of bits that have the value 1 (*1-bits*) in the vector.

The logical operations AND, OR, complementation, and exclusive-OR (XOR) denoted by $u \cdot v$ or uv, $u + v$, $\overline{u}$, and $u \oplus v$, respectively, are performed bitwise on binary vectors. The AND operation forms a *product* and the OR operation forms a *sum.*

2.1.2 Completely Specified Functions

A *completely specified function* of n variables, $f(X), X = \{x_1, x_2, \ldots, x_n\}$, is a mapping from B^n to B, denoted by $f : B^n \rightarrow B$, where $B = \{0, 1\}, x_i \in B$, and B^n is the set of all binary vectors of n bits.

There are 2^n binary vectors in B^n. Thus a switching function may be represented by a 2^n row, $n+1$ column *truth table* which gives an ordered list of the possible binary vectors along with the corresponding values of $f(X)$. If one interprets each binary vector as the binary representation of an integer, the list is specified in ascending numerical order starting with zero. Thus $00\ldots0$ is specified first, then vector $00\ldots01$, and so on until $11\ldots1$ is listed.

Given a standard variable ordering for the truth table, the output values of $f(X)$ may be treated as a binary vector $\mathbf{Z}$ defining $f(X)$. The variable ordering used here is such that variable x_i corresponds to bit u_i of an input vector u.

An alternative column vector $\mathbf{Y}$ defining $f(X)$ is obtained by recoding the binary values $\{0, 1\}$ to the values $\{+1, -1\}$. The usefulness of this coding scheme in the spectral and autocorrelation domains is shown in following sections. Table 2.1 depicts, for each coding, a truth table for an example 3-variable switching function.

Table 2.1: A 3-Variable Function

(a) Using $\{0, 1\}$ coding

k	x_3	x_2	x_1	$f(X)$	**Z**
0	0	0	0	0	0 z_0
1	0	0	1	1	1 z_1
2	0	1	0	1	1 z_2
3	0	1	1	1	1 z_3
4	1	0	0	0	0 z_4
5	1	0	1	1	1 z_5
6	1	1	0	0	0 z_6
7	1	1	1	1	1 z_7

(b) Using $\{+1, -1\}$ coding

k	x_3	x_2	x_1	$f(X)$	**Y**
0	1	1	1	1	1 y_0
1	1	1	−1	−1	−1 y_1
2	1	−1	1	−1	−1 y_2
3	1	−1	−1	−1	−1 y_3
4	−1	1	1	1	1 y_4
5	−1	1	−1	−1	−1 y_5
6	−1	−1	1	1	1 y_6
7	−1	−1	−1	−1	−1 y_7

2.1.3 Incompletely Specified Functions

An *incompletely specified function* is one which is defined for only a subset of the 2^n binary vectors. The unspecified entries are called *don't cares*. Table 2.2 is an example truth table for such a function. Don't care entries are denoted by "–".

A completely specified switching function may be uniquely defined by a truth table; however, an incompletely specified function corresponds to more than one completely specified function. If an incompletely specified function has d don't cares, then there are 2^d possible value assignments for them. Thus an incompletely specified function is a representation of a set of 2^d completely specified functions.

Table 2.2: A 3-Variable Incompletely Specified Function

k	x_3	x_2	x_1	$f(X)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	–
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	1	1	1	0

Table 2.3: A 3-Variable 3-Output Function

k	x_3	x_2	x_1	$f_1(X)$	$f_2(X)$	$f_3(X)$
0	0	0	0	0	0	0
1	0	0	1	1	1	–
2	0	1	0	1	–	1
3	0	1	1	1	1	0
4	1	0	0	0	1	–
5	1	0	1	1	1	1
6	1	1	0	0	0	0
7	1	1	1	1	0	1

2.1.4 Multiple-Output Functions

A set of m functions $f_j(X), X = \{x_1, x_2, \ldots, x_n\}, j = 1, 2, \ldots, m$. is called a *system of m functions*, or, an *m-output function*. An m–output function may be represented by a 2^n row, $n+m$ column, truth table. Each $f_j(X)$ may be either completely specified or incompletely specified. Table 2.3 gives an example of a 3-variable 3-output function.

2.1.5 Switching Expressions and Switching Algebra

Switching functions can be described using *switching expressions* which may be manipulated using the rules of *switching algebra* to perform various tasks. The following briefly discusses switching expressions.

A *literal* is a variable x_i or its complement $\overline{x}_i$. A *product term* is either a literal or a product (AND) of literals; for example, the expression x_3x_2 is a product term. A product term in which each of the n variables of a switching function appears exactly once in either its true or complemented form is called a *minterm*. Thus each of the binary vectors in B^n represents a minterm of $f(X)$. Table 2.4 lists all of the minterms of three variables.

A switching function may be expressed as a *sum of products*. Because several different sum-of-products expressions may exist for a function, the notion of a *canon-*

Table 2.4: General Truth Table for Functions of Three Variables

k	x_3	x_2	x_1	$f(X)$	Minterms	Maxterms
0	0	0	0	c_0	$\overline{x}_3\overline{x}_2\overline{x}_1 = m_0$	$x_3 + x_2 + x_1 = M_0$
1	0	0	1	c_1	$\overline{x}_3\overline{x}_2x_1 = m_1$	$x_3 + x_2 + \overline{x}_1 = M_1$
2	0	1	0	c_2	$\overline{x}_3x_2\overline{x}_1 = m_2$	$x_3 + \overline{x}_2 + x_1 = M_2$
3	0	1	1	c_3	$\overline{x}_3x_2x_1 = m_3$	$x_3 + \overline{x}_2 + \overline{x}_3 = M_3$
4	1	0	0	c_4	$x_3\overline{x}_2\overline{x}_3 = m_4$	$\overline{x}_3 + x_2 + x_1 = M_4$
5	1	0	1	c_5	$x_3\overline{x}_2x_1 = m_5$	$\overline{x}_3 + x_2 + \overline{x}_1 = M_5$
6	1	1	0	c_6	$x_3x_2\overline{x}_1 = m_6$	$\overline{x}_3 + \overline{x}_2 + x_1 = M_6$
7	1	1	1	c_7	$x_3x_2x_1 = m_7$	$\overline{x}_3 + \overline{x}_2 + \overline{x}_1 = M_7$

ical form is utilized to obtain a unique switching expression. Such a canonical form. called a *minterm expansion*, is a sum of minterms in which no two identical minterms appear. It is formed by summing the minterms for which $f(X) = 1$. For example. the function given in Table 2.1 has the minterm expansion:

$$f(X) = \overline{x}_3\overline{x}_2x_1 + \overline{x}_3x_2\overline{x}_1 + \overline{x}_3x_2x_1 + x_3\overline{x}_2x_1 + x_3x_2x_1. \tag{2.1}$$

Minterms are often written in the abbreviated form m_k, where m_k corresponds to row k of the truth table. Thus equation (2.1) may be rewritten as:

$$f(X) = m_1 + m_2 + m_3 + m_5 + m_7 = \sum m(1, 2, 3, 5, 7).$$

In general, for a function of n variables, the minterm expansion may be written as

$$f(X) = \sum_{k=0}^{2^n-1} c_k m_k, \tag{2.2}$$

where each c_k is the value of $f(X)$ corresponding to m_k.

A *sum term* is either a literal or a sum (OR) of literals. A sum term in which each of the n variables of a switching function appears exactly once in either its true or complemented form is called a *maxterm*. Thus each of the vectors in B^n represents a maxterm of $f(X)$. Table 2.4 lists all of the maxterms of three variables.

A switching function may be expressed as a *product of sums*. Again, several different product-of-sums expressions may exist for a function. So a *canonical* form is

used to obtain a unique switching expression. Such a canonical form, called a *maxterm expansion*, is product of maxterms in which no two identical maxterms appear. It is formed by multiplying (ANDing) the maxterms for which $f(X) = 0$. For example, the function given in Table 2.1 has the maxterm expansion:

$$f(X) = (x_3 + x_2 + x_1)(\overline{x}_3 + x_2 + x_1)(\overline{x}_3 + \overline{x}_2 + x_1). \tag{2.3}$$

Maxterms are often written in the abbreviated form M_k, where M_k corresponds to row i of the truth table. Thus equation (2.3) may be rewritten as:

$$f(X) = M_0 M_4 M_6 = \prod M(0, 4, 6).$$

In general, the maxterm expansion for a function of n variables may be written as

$$f(X) = \prod_{k=0}^{2^n - 1} (c_k + M_k), \tag{2.4}$$

where each c_k is the value of $f(X)$ corresponding to M_k.

A switching function may be described by several different but equivalent expressions. Two examples of such expressions are a function's minterm expansion and maxterm expansion.

A switching circuit implementing a switching function has a direct correspondence with a switching expression for the function. Thus the complexity or cost of a switching circuit is related to the complexity of the expression describing the function. Therefore to minimize the cost of a switching circuit realizing a function, one may simplify or *minimize* a switching expression using the rules of switching algebra. The process of minimizing a switching expression is called *Boolean minimization.*

The definition of a *minimal expression* or the measure of "goodness" that is used to select or construct a desirable expression must take into consideration some of the characteristics of the physical switching circuit to be used. An expression that is desirable for one type of circuit may not be a good choice for another. Two common criteria that are used to define a minimal expression are:

1. A minimal expression is the one that has a minimum number of literals in it. Under this rule an expression is considered minimal if no other expression has fewer literals in it.

2. A minimal expression is a sum-of-products (product-of-sums) expression with a minimum number of terms in it. Here, an expression is deemed minimal if it has the fewest product (sum) terms unless another expression exists with the fewest terms but also has fewer literals.

Given a definition of a switching function, the goal of circuit synthesis procedures is to find a minimal expression that can be mapped to the underlying physical switching elements. For functions with small numbers of inputs, one can manually construct minimal expressions using a variety of methods. Some of these methods include algebraic manipulation techniques and geometric techniques such as the *Karnaugh map* method [43]. However, most useful functions have numbers of inputs that make manual procedures impractical, if not impossible. Additionally, most useful functions are multiple-output functions and may also be incompletely specified. These factors greatly complicate minimization procedures.

Therefore, various automated systematic minimization procedures have been developed [13, 14, 15, 23, 36, 82]. Even these procedures, due to the exponential growth in the size of a function with respect to the number of inputs (there are 2^n binary vectors defining $f(X)$), may not arrive at a minimal expression and will resort to constructing a *near-minimal* or "good" expression based on a set of heuristics.

2.1.6 Representation of Switching Expressions

There are a number of constructs that may be used to represent switching functions and switching expressions. The truth table and Karnaugh map [43] are two such constructs.

Another construct is the *partition matrix*, which is a tabular representation of a switching function $f(X)$, $X = \{x_1, x_2, \ldots, x_n\}$. Let X be partitioned into two subsets X_1 and X_2, $X_1 \cap X_2 = \phi, X_1 \cup X_2 = X$ and let $s = |X_1|$. Then the partition matrix of $f(X)$ has 2^{n-s} rows and 2^s columns. The columns are labeled with the 2^s assignments to the variables in X_1 and the rows are labeled with the 2^{n-s} assignments to the variables in X_2. The entries in the matrix are the minterms of the function.

However, all of these representations require 2^n entries for an n-variable function. Thus for even relatively small values of n, they are quite large and unmanageable. To represent switching functions more compactly, various other constructs may be used.

Cube Lists

A commonly used construct to define functions is the *cube list*. The procedures developed in this dissertation all utilize this construct to represent and manipulate the functions they process.

A *cube* is an n-tuple of elements from $\{0, 1, -\}$ representing some subset of the n-variable minterms. The minterms included in this subset are those formed by substituting 0's and 1's for the "$-$"s in all possible ways. This subset is called the *cover* of the cube. For example, the cube "$-01-$" represents the set of minterms $\{0010, 0011, 1010, 1011\}$.

A minterm is *covered* by a cube if there is a 0 or 1 assignment to the "$-$"s in the cube that yields that minterm. A cube that contains k "$-$"s covers 2^k minterms and a cube containing no "$-$"s represents a single minterm. Each cube may cover only minterms for which a function has the same value for all minterms in the cover.

The *intersection* of two cubes c_i and c_j, denoted as $c_i \cap c_j$, is the cube representing the set of minterms that are covered by both c_i and c_j. If there are no minterms in common, then c_i and c_j are said to be *disjoint*. A *disjoint cover* of a function is a

cube list in which all the cubes are mutually disjoint.

The *sharp* [27] operation on cubes c_i and c_j, denoted as $c_i \# c_j$, represents all minterms that are covered by c_i but not by c_j. The cube representation of the result may have more than one cube. The *disjoint sharp* [27] operation is similar except that the result must be expressed as disjoint cubes. A disjoint cover of a function may be obtained from a non-disjoint cover by application of the disjoint sharp operation on cubes that have a non-empty intersection.

In a cube list, both the cube and the value that the function has for the cube are listed. Since the cube is specifying a set of input assignments to a function's input variables, the cube is also referred to as the *input part* of the entry. The function output value is termed the *output part.*

The term "cube" is used to refer to both the set of minterms that are covered as well as to a cube list entry. For example, the cube lists that represent the functions given in Table 2.1 and Table 2.2 are

```
--1  1          000  0
-00  0          010  -
01-  1   and    0-1  1 ,
110  0          10-  1
                11-  0
```

respectively.

Multiple-output functions may also be defined with cube lists. For this purpose the output part of each cube contains m values, one for each of the m outputs. To aid in the definition of multiple outputs another symbol, "$\sim$", is used to indicate that the cube is not defining any minterms for the corresponding function. For example, a cube list defining the multiple output function given in Table 2.3 is

```
000  000
0-1  11~
1-1  1~1
11-  ~0~
10-  ~1~
1-0  0~~ .
010  1-1
110  000
001  11-
100  01-
011  110
```

Sum-of-products expressions for a system of switching functions may be represented by a cube list, in which case each cube represents a product term of the expression. In the context of a product term a "–" in the input part of a cube indicates that the product term is independent of the corresponding input variable, i.e. the variable does not appear as a literal in the term.

A cube list may be divided into sublists, each containing cubes that define only one value. One sublist would define only the 0's of the functions and another would define only the 1's. For an incompletely specified function, a third list would define the don't cares. If a cube list is divided in this manner, one of the sublists may be eliminated and any minterm that is not explicitly defined by a cube would assume the value the eliminated list would otherwise have specified for it. This can greatly reduce the size of a cube list.

Cube lists are produced by many of the available minimization procedures [13, 14, 15, 82]. For example the cube list produced by Espresso [13] for the multiple-output function defined in Table 2.3 is provided in Figure 2.1c. These correspond to the sum-of-products expressions listed in Figure 2.1b. Note that the cube list of Figure 2.1c only specifies the minterms that take the value 1. The remaining minterms are assumed to be 0.

2.1.7 Switching Circuit Structures

There are many physical circuit structures that are used to implement switching functions. The following is a short list of some of the these structures.

- random logic
- programmable logic arrays
- gate arrays
- field programmable gate arrays
- standard cells

However, only those structures that have some significance to the research presented in this dissertation are discussed.

Chapter 3 deals with the optimization of Programmable Logic Arrays (PLAs): however, the developed optimization techniques are also applicable to synthesis techniques aimed at Field Programmable Gate Arrays (FPGAs).

The methods discussed in Chapters 5 through 7 are applicable to FPGAs as well as to random logic implementations of switching circuits.

Programmable Logic Arrays

The cost of a circuit using AND and OR gates to realize a function is directly related to the number of gates and the number of gate inputs that are required. A two-level circuit consists of a set of AND gates providing the inputs to a single OR gate. A switching expression that is considered minimal according to criterion (2) corresponds directly to a two-level switching circuit with minimum number of gates and minimum number of gate inputs.

A *Programmable Logic Array* or *PLA* is a regularly structured two-level AND/OR circuit that realizes directly a set of sum-of-products expressions describing a set of switching functions. Efficient Boolean minimization methods [13, 23, 36, 82] have been developed for determining minimal or near-minimal expressions that are targeted to PLA implementation.

This dissertation concerns itself only with the logical structure of a PLA and not with any of the various technology dependent implementations of PLAs. In addition, only the *core* of a PLA is considered in this dissertation for purposes of algorithm evaluation.

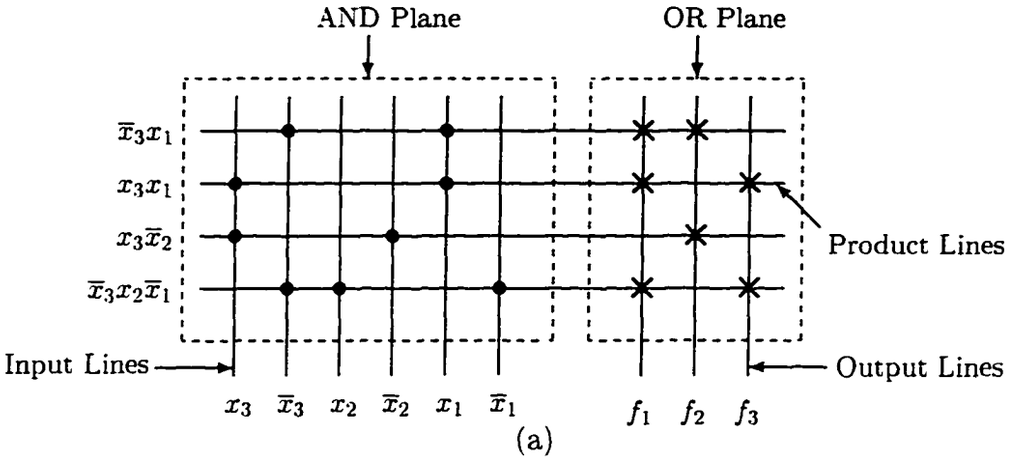
The core of a PLA is logically structured into an AND plane, which implements the product terms, and an OR plane, which forms the sums, each arranged as an array of intersecting *lines*. A *crosspoint* may be placed at any site where two lines intersect. This serves as a selection mechanism.

The AND plane consists of a set of *input lines*, which form the literals, and a perpendicular set of *product lines*, which form the product terms. There are $2n$ input lines which carry the values of the variables and their complements. These values are generated to the core of the PLA by one-bit decoders. Each product term requires one product line. The literals for each product term are selected by placing crosspoints at the sites where the appropriate input lines intersect with the corresponding product line.

The OR plane consists of a set of *output lines* running perpendicular to the product lines. There is one output line for each output of the system of functions that is implemented. The sum of products for a function output is formed by placing crosspoints on the output line where the appropriate product lines intersect.

Figure 2.1a depicts an example PLA that implements the set of functions listed in Figure 2.1b. These functions represent one way of assigning 0 or 1 to the don't cares of the function given in Table 2.3. For ease of reference each product term is

Figure 2.1: PLA Implementing Example Multiple-output Function



(a)

$$f_1(X) = \overline{x}_3x_1 + x_3x_1 + \overline{x}_3x_2\overline{x}_1$$
$$f_2(X) = \overline{x}_3x_1 + x_3\overline{x}_2$$
$$f_3(X) = x_3x_1 + \overline{x}_3x_2\overline{x}_1$$

(b)

```
0-1  11~
1-1  1~1
10-  ~1~
010  1~1
```

(c)

displayed next to the corresponding product line. Large dots represent crosspoints in the AND plane and ×'s represent crosspoints in the OR plane.

Field Programmable Gate Arrays

Recently, interest has grown in the use of *Field Programmable Gate Arrays* or *FPGAs*. An FPGA is a circuit structure that is composed of a regularly arranged set of *configurable logic blocks* (CLBs) and programmable interconnect to establish connections between the various elements on the chip. The CLBs implement the logic and generally contain memory elements for storage of data. FPGAs also contain input and output elements on the periphery to interface the FPGA to the external environment. A switching function is implemented in an FPGA by programming the appropriate subfunctions into the logic blocks and establishing the appropriate connections be-

tween logic blocks and other elements on the chip. Unlike in a PLA, more than two levels are typically used to implement switching functions in an FPGA.

The main difference between FPGAs and other *gate array* devices is that FPGAs are programmable by the end user using a relatively inexpensive set of tools. This enables an implementor to create a circuit on a chip without having to take the high-cost and slow path of having a full or semi-custom chip manufactured.

Therefore a circuit may be implemented quickly and at a relatively low cost. These are important advantages for prototyping and product development. Additionally, because of the relatively low development cost, FPGAs are now used in many practical applications.

A disadvantage is that FPGAs do not achieve the density and speed of a full or semi-custom design. However, FPGAs are continually improving in both respects.

There are four basic architectures used in FPGAs to implement switching circuits:

1. **PLD-based architectures.** The FPGAs consist of a small number of *Programmable Logic Devices* (PLDs), which implement two-level circuitry in a fashion similar to that of a PLA. This type of FPGA, sometimes called a complex PLD or CPLD is manufactured by Altera [6] and AMD [4].

2. **Sea-of-gates architectures.** An FPGA consists of an array of logic blocks, each of which usually consists of a small gate network. Examples are the Motorola MPA1000 [71] and the Xilinx XC8100 [99] families of fine-grained FPGAs.

3. **Symmetrical array architectures.** These FPGAs consist of logic blocks that are arranged as an array with interconnect that runs between the rows and columns for the full length and width of the chip. An example is Xilinx's XC4000 [98] family of FPGAs.

4. **Row based architectures.** The logic blocks are arranged into rows with interconnect running between the rows. Logic blocks are connected together via connection to the inter-row interconnect using vertical interconnect segments. Texas Instruments [88] and Actel [2, 3] manufacture this type of FPGA.

The logic blocks may be implemented using several structures. The most common are:

1. **Basic gates.** Each logic block consists of a single logic gate such as AND, NAND, XOR etc., or a small network of logic gates. Motorola's MPA1000 [71] and Xilinx's XC8100 [99] families of fine-grained FPGAs use this type of logic block

2. **Lookup-Tables (LUTs).** The logic blocks consist of LUTs. A subfunction is implemented as a look-up table and the subfunction inputs serve as the index into the LUT. Xilinx's XC3000 and XC4000 [98] FPGAs use LUTs for logic blocks.

3. **PLDs.** The logic blocks consist of PLAs, PALs or combinations of these.

4. **Multiplexors.** Texas Instruments' TPC [88] series of FPGAs, and Actel's ACT [2, 3] family of FPGAs are examples that use multiplexors for logic blocks.

There are several mechanisms that are used to "program" an FPGA. These are divided into two main categories: program-once and re-programmable mechanisms. Re-programmable FPGAs can be reconfigured whereas program-once FPGAs cannot.

Since more than two levels of circuitry are typically used in an FPGA and there are different techniques employed to implement subfunctions, not only is the definition of a minimum expression for a switching function different from that for a PLA, but also depends on the type of FPGA to be used to implement the function.

For example, for a n-input LUT-based FPGA, a function that has n or fewer inputs can be implemented with no minimization since its entire truth table can be contained completely within a single logic block. However, for a Motorola MPA1000 FPGA, the same n-input function would need to be minimized so that a minimum number of two-input structures is required.

Measuring the complexity of a switching circuit implementation typically takes into account the factors of area, speed, and power consumption and is thus highly technology dependent. Often simple estimators are used as a first approximation of complexity. These include: the number of product terms in a PLA; or the number of gates and the number of gating levels, or the number of transistors in a multi-level realization. Since this dissertation deals with differing implementation strategies, different complexity measures are used as appropriate. The measures are explained as they are introduced. The most complex measure, the number of transistors in a CMOS gate level implementation, is detailed in Section 5.4.1.

The work described in Chapter 3 is applicable to PLD-based FPGA structures whereas the work in Chapters 5 through 7 is applicable to multiplexor and sea-of-gates-based FPGAs. The work in this dissertation is not directly aimed at LUT-based architectures, but, the circuits generated can be re-targeted to LUT-based FPGAs using suitable technology mapping methods [31, 14].

2.2 The Spectral Domain

In the *spectral* representation, a function is described by a *spectrum*. Like the truth table of the Boolean or functional domain, the spectrum contains 2^n values. However, unlike in the Boolean domain, each value of the spectrum contains some information about the global behaviour of the function at all 2^n input assignments. In addition, the values in the spectral domain are integers and not Boolean values.

The procedures developed in this dissertation compute the autocorrelation from a function's spectrum more efficiently than from the function's truth table. The procedures also take advantage of knowledge regarding the spectra of switching functions. For example, spectral invariance operations, as applied to the autocorrelation. are used in Chapters 5 and 6 in enhancements to a particular multi-level synthesis approach called two-place decomposition. In Chapter 7 knowledge of the sizes of spectral equivalence classes is used in the development of a multi-level synthesis approach aimed at two-place decomposition.

2.2.1 The Hadamard Transform

The conversion of $f(X)$ from the Boolean domain into the spectral domain is accomplished using the *Hadamard* transform [39, 48]. The spectrum for a function may be computed for $f(X)$ using either the $\{0, 1\}$ or the $\{+1, -1\}$ coding.

Let $\mathbf{R}$ denote the spectrum computed for $f(X)$ from $\mathbf{Z}$ and let $\mathbf{S}$ denote the spectrum computed for $f(X)$ from $\mathbf{Y}$. $\mathbf{R}$ and $\mathbf{S}$ are given by

$$\mathbf{R} = \mathbf{T}^n\mathbf{Z}, \quad \text{and} \quad \mathbf{S} = \mathbf{T}^n\mathbf{Y}, \tag{2.5}$$

where $\mathbf{T}^n$ is a Hadamard transform matrix [39, 48], a complete orthogonal square matrix of dimension $2^n \times 2^n$, in which the rows are the Rademacher-Walsh functions [39, 48]. $\mathbf{T}^n$ has the following recursive structure:

$$\mathbf{T}^0 = [1], \qquad \mathbf{T}^n = \begin{bmatrix} \mathbf{T}^{n-1} & \mathbf{T}^{n-1} \\ \mathbf{T}^{n-1} & -\mathbf{T}^{n-1} \end{bmatrix}.$$

For example:

$$\mathbf{T}^1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

$$\mathbf{T}^2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}.$$

Each row of $\mathbf{T}^n$ is one of the Rademacher-Walsh functions [39, 48]. These functions are the set of n-variable linear (*Exclusive OR* or XOR) functions coded in $\{+1, -1\}$. There are 2^n possible XOR functions of n variables. Figure 2.2 gives the XOR functions represented by the rows of $\mathbf{T}^3$. Since the variable ordering as described in the previous section is used to define $\mathbf{Z}$ (or $\mathbf{Y}$), the variables involved in the XOR function are those that correspond to the 1-bits in the binary representation of the row number of $\mathbf{T}^n$, where the rows are numbered starting with zero.

Figure 2.2: Rademacher-Walsh Functions of $\mathbf{T}^3$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix} \begin{matrix} 0 \\ x_1 \\ x_2 \\ x_1 \oplus x_2 \\ x_3 \\ x_1 \oplus x_3 \\ x_2 \oplus x_3 \\ x_1 \oplus x_2 \oplus x_3 \end{matrix}$$

Each element of $\mathbf{R}$ (or $\mathbf{S}$) is called a *spectral coefficient* and is identified by the variables involved in the corresponding XOR function. The *order* of each coefficient is given by the number of variables involved in the corresponding XOR function. Equivalently, each coefficient may be identified by its numeric row position in the spectrum and its order is given by the weight of the binary representation of that

position. For example the spectral coefficient at row position 6 corresponds to the XOR function $x_2 \oplus x_3$.

The following example spectral computation uses the function given in Table 2.1. Each spectral coefficient is identified by the variables involved in the corresponding XOR function. For the $\{0,1\}$ coding, $\mathbf{Z}$ is transformed into $\mathbf{R}$ using $\mathbf{T}^3$.

$$\begin{array}{cccc} \mathbf{T}^3 & & \mathbf{Z} & = & \mathbf{R} & \\ \left[\begin{array}{cccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{array}\right] & \left[\begin{array}{c} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{array}\right] & = & \left[\begin{array}{r} 5 \\ -3 \\ -1 \\ -1 \\ 1 \\ 1 \\ -1 \\ -1 \end{array}\right] & \begin{array}{l} r_0 \\ r_1 \\ r_2 \\ r_{12} \\ r_3 \\ r_{13} \\ r_{23} \\ r_{123} \end{array} \end{array}$$

For the $\{+1,-1\}$ coding, $\mathbf{Y}$ is transformed into $\mathbf{S}$ using $\mathbf{T}^3$ as follows

$$\begin{array}{cccc} \mathbf{T}^3 & & \mathbf{Y} & = & \mathbf{S} & \\ \left[\begin{array}{cccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{array}\right] & \left[\begin{array}{r} 1 \\ -1 \\ -1 \\ -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{array}\right] & = & \left[\begin{array}{r} -2 \\ 6 \\ 2 \\ 2 \\ -2 \\ -2 \\ 2 \\ 2 \end{array}\right] & \begin{array}{l} s_0 \\ s_1 \\ s_2 \\ s_{12} \\ s_3 \\ s_{13} \\ s_{23} \\ s_{123} \end{array} \end{array}$$

In the above examples, the spectral coefficients are identified by subscripts specifying the variables involved in the corresponding exclusive-OR function. That notation

is used in specified instances where it provides the clearest explanation. However, throughout most of the development it is convenient to identify each coefficient by a decimal subscript, the binary representation of which has 1's in the positions corresponding to the variables involved in the relevant exclusive-OR function. This notation is used throughout the dissertation unless otherwise noted.

No information is lost in the above transformation. The information that is contained in the spectral domain is the same as that contained in the Boolean domain. The data of each domain can be generated uniquely from the data in the other. The inverse transform is given by

$$\mathbf{Z} = \tfrac{1}{2^n}\mathbf{T}^n\mathbf{R}, \quad \text{and} \quad \mathbf{Y} = \tfrac{1}{2^n}\mathbf{T}^n\mathbf{S}. \tag{2.6}$$

Additionally, the spectral coefficients of $\mathbf{R}$ contain the same information as the coefficients of $\mathbf{S}$. The two spectra are linearly related as follows:

$$r_0 = \tfrac{1}{2}(2^n - s_0), \qquad r_i = -\tfrac{1}{2}s_i, i \neq 0$$

Each spectral coefficient provides a measure of correlation between the given function output vector for $f(X)$ and the corresponding Rademacher-Walsh function. The magnitude of a coefficient value indicates the degree of similarity and the sign indicates whether $f(X)$ is more similar to the XOR function or to the complement of the XOR function (XNOR). A coefficient with maximum magnitude indicates perfect correlation (i.e. $f(X)$ is equal to the corresponding XOR function).

Let $E(X)$ = a Rademacher-Walsh function. Each $\mathbf{R}$ coefficient value equals $\|\overline{E}(X)f(X)\| - \|E(X)f(X)\|$. That is, the number of places $f(X)$ is equal to $\overline{E}(X)$ minus the number of places where $f(X)$ is equal to $E(X)$. Each $\mathbf{S}$ coefficient value equals the number of places at which $E(X)$ and $f(X)$ agree minus the number of places that $E(X)$ and $f(X)$ disagree.

2.2.2 Spectrum Computation

There is a cost associated with the transformation of $f(X)$ from the Boolean domain to the spectral domain. Computation of the spectrum using Equation 2.5 requires on the order of 2^{2n} operations. As n grows, this quickly becomes infeasible. Due to the recursive nature of $\mathbf{T}^n$ a *Fast Hadamard Transform* [39] may be used to compute the spectrum using on the order of $n2^n$ operations. Although this is much better, the number of operations as well as memory requirements still grows exponentially with n. This presents difficulties when trying to deal with functions with large numbers of inputs.

However, there are situations in which the full spectrum is not required. In these cases individual or small subsets of the coefficients may be calculated efficiently from cube lists [72, 91, 93].

2.2.3 Handling Incompletely Specified Functions

Incompletely specified functions present another challenge to the use of the spectrum in switching function analysis and circuit synthesis.

$\mathbf{S}$ may be computed for incompletely specified functions by extending the encoding $\{+1, -1\}$ to $\{+1, 0, -1\}$. In this coding scheme 0 is used to represent a don't-care and the other values are used as described earlier. Difficulty arises in the interpretation of the resulting spectrum. The coefficients could be viewed to have the average value over all possible 0 and 1 assignments to the don't cares. Alternatively, the don't cares can be viewed as being ignored. In either case, the don't cares do not contribute to the information content of the coefficients.

Value assignments may be made to the don't cares. There are several options in this regard. They include assigning a random value to each don't care, arbitrarily assigning all don't cares the same value (0 or 1), or attempting to assign values to

the don't cares such that some desired functional behaviour is achieved. This last alternative can be too computationally expensive to perform.

A method described in [39] utilizes two vectors to describe an incompletely specified function: one in which all don't cares are assigned the value 0, and one in which the don't cares are assigned the value 1. Spectra are computed for the two vectors and relative values between the coefficients of the two spectra are used to determine characteristics of the function.

2.2.4 Applications of Spectra

Each spectral coefficient contains some information about the global behaviour of the function at all 2^n input assignments. Thus some characteristics and properties may be identified more easily from the spectrum than from the function's truth table. Moreover, some operations or transformations that are to be performed on the function can be done much more readily in the spectral domain.

The following subsections discuss the use of the spectrum of the function to identify functional characteristics and to perform operations that are more difficult to do in the functional domain.

Function Complexity Estimate

Let $L(f)$ be the minimal number of one and two-input gates in a minimal switching circuit realizing $f(X)$. It is generally accepted that to compute $L(f)$ requires roughly the same amount of effort as to determine a minimal realization for $f(X)$. So complexity criteria are used that are easily computed and produce good estimates of the complexity of a function.

One measure that has been shown by experiment to provide a satisfactory estimate

of $L(f)$ is [38, 48]

$$C(f) = \sum_{\|u\|=1} \sum_{v=0}^{2^n-1} (f(v)f(v \oplus u) + \overline{f}(v)\overline{f}(v \oplus u)). \tag{2.7}$$

A pair of input assignments u and v are considered *adjacent* if $d(u, v) = 1$. Equation 2.7 counts of the number of adjacent pairs of assignments for which $f(X)$ takes the same value for both assignments in the pair [48]. A higher adjacency count suggests that the function has a lower complexity.

In the spectral domain $C(f)$ is computed as

$$C(f) = n2^n - \frac{1}{2^{n-2}} \sum_{v=0}^{2^n-1} \|v\| r_v^2, \tag{2.8}$$

where $\|v\|$ is the number of 1's in the binary representation of v.

Equation 2.8 is a weighted sum of the spectral coefficients where the weight is the order of the coefficient. Thus $C(f)$ has a larger value, and therefore $f(X)$ has a smaller estimated complexity, if the lower order coefficients have large magnitudes.

This intuitively follows from the interpretation of the spectral coefficients as measures of correlation between $f(X)$ and the Rademacher-Walsh functions. The higher the order of the coefficient the greater the number of variables involved in the corresponding XOR function and thus the more complex $f(X)$ should be.

Spectral Invariance Operations

In presenting the invariance operations, the spectral coefficients are identified by subscripts which consist of the input-variable numbers that are involved in their calculation. The symbol α refers to subsets of the input variable numbers including ϕ.

There are five invariance operations [28, 37, 39, 48] that may be applied to the spectral coefficients. The invariance operations perform only a permutation of the

coefficients within the spectrum and the magnitudes of the complete set of coefficients remain invariant. The invariance operations are:

1. Complementation of any input variable x_i. This negates 2^{n-1} spectral coefficients: $s_{i\alpha} \rightarrow -s_{i\alpha}, \forall\alpha,\ i \notin \alpha$.

2. Permutation of any input variables x_i and $x_j, i \neq j$. This exchanges 2^{n-2} pairs of coefficients: $s_{i\alpha} \leftrightarrow s_{j\alpha} \forall\alpha,\ i,j \notin \alpha$.

3. Complementation of the function output. This requires the negation of all 2^n coefficients: $s_\alpha \rightarrow -s_\alpha, \forall\alpha$.

4. Replacement of any variable x_i with $x_i \oplus x_j, i \neq j$. This interchanges 2^{n-2} pairs of coefficients: $s_{i\alpha} \leftrightarrow s_{ij\alpha}, \forall\alpha,\ i,j \notin \alpha$.

5. Replacement of $f(X)$ by $f(X) \oplus x_i$. This results in the interchange of 2^{n-1} pairs of spectral coefficients: $s_{i\alpha} \leftrightarrow s_\alpha, \forall\alpha, i \notin \alpha$.

Although the operations are described relative to $\mathbf{S}$, the results of performing the operations on $\mathbf{R}$ are identical except that r_0 is handled as follows:

- for Type 3 translations r_0 is replaced by $2^n - r_0$
- for Type 5 translations r_0 is replaced by $2^{n-1} + r_i$ and r_i is replaced by $r_0 - 2^{n-1}$

Operations 1 through 3 do not alter the order of any of the coefficients. The remaining two are *linear translations* that are performed easily in the spectral domain. Operation 4 exchanges coefficients from all orders except s_0 and operation 5 includes s_0 in the interchange. Thus by using type 4 and type 5 translations, any coefficient may be moved to any order.

Linearization

Karpovsky [48] has shown that a *linearization procedure* employing the invariant spectral operations may be used to move the largest-magnitude spectral coefficients into the first-order positions. This results in a spectrum that maximizes the complexity estimate $C(f)$. Thus the linearization procedure transforms the function $f(X)$ into a simpler function $g(X)$.

The realization of $f(X)$ consists of a linear prefilter implementing the operations performed in the linearization procedure, followed by the simpler $g(X)$.

Details of the linearization procedure are discussed in Chapter 6.

Classification of functions

Function classification is the process of dividing the 2^{2^n} possible functions that may be defined over n input variables into groups or *equivalence classes* based on certain common functional properties. This has several advantages.

First, all 2^{2^n} functions of $\leq n$ variables may be classified in some compact manner since only one function from each class needs to be represented. From a functional analysis viewpoint this greatly reduces the amount of work involved in analyzing the functions.

Second, each of the functions in a particular class may be realized with a single switching circuit augmented with circuitry to perform the operations corresponding to the classification procedure.

The NPN classification [34, 37] divides the 2^{2^n} possible functions of $\leq n$ variables into equivalence classes. For any pair of functions in an equivalence class, one function can be obtained from the other by applying any subset of the following three operations:

Table 2.5: Classes for Functions of $\leq n$ Variables
— - Value not computed

Classification Method	Number n of input variables					
	1	2	3	4	5	6
Total number of functions	4	16	256	65536	$\simeq 4.3 \times 10^9$	$\simeq 1.8 \times 10^{19}$
NPN-equivalent classes	2	4	14	222	616126	$\simeq 2 \times 10^{14}$
Spectral-equivalent classes	1	2	3	8	48	—

1. negation (N) (complementation) of one or more of the function's input variables x_i;

2. permutation (P) of two or more of the function's input variables x_i;

3. negation (N) (complementation) of the output of the function.

The number of NPN classes of the possible functions of $\leq n$ variables is much less than 2^{2^n} but still grows extremely quickly with n, as can be seen in Table 2.5 [37, 39].

Spectral classification methods further reduce the number of classes by using the spectral invariance operations. This is possible since, in addition to the NPN operations, the two linear translation operations are provided. The numbers of spectral classes for functions of $\leq$ n variables are given in Table 2.5. As shown in the table, the number of spectral-equivalent classes is reduced tremendously from the number of NPN classes.

The classification procedure yields a representative or *canonical* function for a class. The canonical function for a spectral class is ascertained by performing a linearization on the spectrum followed by a Type 5 linear translation if necessary.

All of the functions in a particular class exhibit the same functional properties. This has implications not only for functional analysis but also for synthesis. Any procedure that takes advantage of certain properties to perform synthesis will work

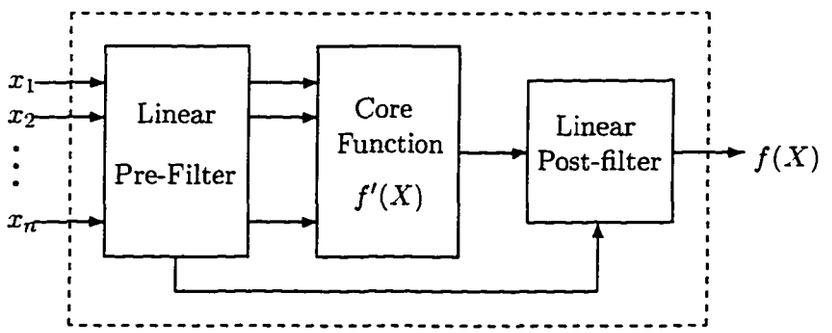
on any function in a class. That is, if one function in a class can be successfully synthesized by a synthesis method, then every function in that class can be synthesized by that method.

Synthesis by Spectral Translation

The following describes a possible circuit synthesis procedure using the spectrum of a function. Given any function, the spectrum may be transformed to yield the canonical spectrum for the class to which the function belongs.

The function then can be implemented as depicted in Figure 2.3. The *core* function is the circuit realizing the canonical function for the class. The *pre-filter* and *post-filter* are circuits that perform the translations corresponding to the classification procedure. The pre-filter corresponds to the linearization of the spectrum and the post-filter corresponds to any Type 5 spectral translation that is needed.

Figure 2.3: Spectral Synthesis Approach



2.3 The Autocorrelation Function

For a Boolean function $f(X)$, $X = \{x_1, x_2, \ldots, x_n\}$ of n variables, the autocorrelation function $\mathbf{B}(u)$ [48] is given by

$$\mathbf{B}(u) = \sum_{v=0}^{2^n-1} f(v)f(v \oplus u), \tag{2.9}$$

where u and v are binary vectors.

$\mathbf{B}(u)$ is a measure of correlation between the function $f(X)$ and $f(X)$ "shifted" u places. The result of the shift of $f(X)$ is a permutation of $\mathbf{Z}$, accomplished by switching element v in $\mathbf{Z}$ with element $v \oplus u$.

From another viewpoint, if $\mathbf{Z}$ is restructured into a partition matrix, as defined in Section 2.1, with columns defined by variables x_i where $u_i = 1$, then from (2.9) the value of $\mathbf{B}(0)$ is the number of 1's in the function and the value of $\mathbf{B}(u)$ is a measure of similarity between pairs of columns j and $j \oplus u$, $j = 0, 1, \ldots, 2^{\|u\|} - 1$. Note that columns j and $j \oplus u$ are at distance $\|u\|$ from each other with respect to variables x_i, where $u_i = 1$.

For a system of m functions $f_i(X)$, $i = 1, \ldots, m$, the autocorrelation function definition is extended to the *total autocorrelation function* [48]

$$\mathbf{B}(u) = \sum_{i=1}^{m} \mathbf{B}_i(u) = \sum_{i=1}^{m} \sum_{v=0}^{2^n-1} f_i(v)f_i(v \oplus u). \tag{2.10}$$

Let $\mathbf{B}$ be a column vector containing 2^n elements $b_u, u = 0, 1, 2, \ldots, 2^{n-1}$, where $b_u = \mathbf{B}(u)$. Each element b_u is called a *total autocorrelation coefficient* and has *order* $\|u\|$. As for spectral coefficients, each autocorrelation coefficient may be identified by a subscript listing the numbers of the variables with which it is associated.

The procedures developed in this dissertation utilize various characteristics of the autocorrelation to perform optimization and synthesis of combinational logic. In particular the PLA optimization methods in Chapter 3 identify near-optimal pairings

for decoded PLAs by exploiting the fact that the autocorrelation coefficients contain measures of similarity within the function.

Chapter 5 takes advantage of the type 4 linear translation in the mappings for a particular type of two-place decomposition. Chapter 6 uses linearization to extend the two-place decomposition method to handle functions that are non-two-place decomposable.

Chapter 7 uses the knowledge of the number of spectral equivalence classes of the possible functions of $\leq n$ variables to place effective constraints on a multi-level synthesis method aimed at two-place decomposition.

2.3.1 Autocorrelation Computation

The computation of $\mathbf{B}$ using equation (2.10) requires $O(m2^{2n})$ operations. To reduce the computational complexity, $\mathbf{B}$ may be computed from the spectra of the system of Boolean functions as follows.

Each $\mathbf{B}_i$ is computed from the spectrum of f_i [48]:

$$\mathbf{B}_i = \frac{1}{2^n}\mathbf{T}^n\mathbf{R}_i^2, \tag{2.11}$$

where the notation $\mathbf{R}^2$ denotes the vector

$$(r_0^2, r_1^2, \ldots, r_{2^n-1}^2).$$

Using equations (2.5) and (2.11), the computation of $\mathbf{B}$ requires $O(mn2^n)$ operations when a fast transform procedure [39] is used.

Table 2.6 exemplifies the computation of $\mathbf{B}$. It provides a system of four-variable functions, $f_i(X), i = 1, 2, 3$, the $\mathbf{R}_i$, and $\mathbf{B}_i$ for each f_i, and the resultant $\mathbf{B}$.

The autocorrelation may also be computed using the $\{+1, -1\}$ coding. In this case each autocorrelation coefficient provides the number of matches minus the number

Table 2.6: Example of Autocorrelation Computation

v,u	f_1	f_2	f_3	$\mathbf{R}_1$	$\mathbf{R}_2$	$\mathbf{R}_3$	$\mathbf{B}_1$	$\mathbf{B}_2$	$\mathbf{B}_3$	$\mathbf{B}$
0000	1	0	1	7	7	8	7	7	8	22
0001	0	0	0	-1	-1	0	2	4	0	6
0010	0	0	0	-1	-1	0	2	4	0	6
0011	1	0	1	-1	3	4	4	4	8	16
0100	0	1	1	-1	-1	0	2	2	4	8
0101	1	1	0	3	3	0	4	2	4	10
0110	0	0	0	-1	-1	0	2	0	4	6
0111	0	1	1	3	-1	4	2	2	4	8
1000	0	1	1	-1	-1	0	2	2	4	8
1001	0	0	0	-1	-1	0	2	0	4	6
1010	1	1	0	3	3	0	4	2	4	10
1011	0	1	1	3	-1	4	2	2	4	8
1100	0	0	0	3	-5	0	2	4	4	10
1101	1	0	1	-1	-1	0	4	4	4	12
1110	1	0	1	-1	-1	0	4	4	4	12
1111	1	1	0	3	-1	-4	4	6	4	14

of mismatches between function values at Hamming distance $\|u\|$ with respect to the variables x_i, where $u_i = 1$. Let $\mathbf{C}$ be the autocorrelation computed from $f(X)$ encoded using the $\{+1, -1\}$ coding. $\mathbf{C}$ may be computed from $\mathbf{S}$ as follows:

$$\mathbf{C} = \frac{1}{2^n}\mathbf{T}^n\mathbf{S}^2. \tag{2.12}$$

$\mathbf{B}$ and $\mathbf{C}$ are linearly related similar to $\mathbf{R}$ and $\mathbf{S}$.

As with the computation of the spectrum, the computation of the autocorrelation, even with the fast transform, requires computation and storage that grows exponentially in n. If only a subset of the coefficients are required, then more efficient algorithms may be used to compute them. For example the first and second order coefficients may be computed very efficiently from a cube list [61]. The algorithm requires on the order of C^2 operations, where C is the number of cubes in the cube list describing the system of functions.

2.3.2 Incompletely Specified Function Handling

If $\{+1, 0, -1\}$ is used, then **C** may be computed for incompletely specified functions. Here the contributions of the don't cares can be interpreted in the same manner as in the spectral computations for incompletely specified functions.

Karpovsky has suggested the use of a *cross correlation* [48] of a pair of recoded completely specified functions to describe an incompletely specified function [49]: one with 1's where there are 1's in the original function, and one in which there are 1's where there are 0's in the original function. The cross correlation of these two functions is used to determine characteristics of the original incompletely-specified function.

2.3.3 Applications of Autocorrelation

Like the spectral coefficients, the autocorrelation coefficients contain some global information regarding all 2^n valuess of a function and may be used to determine properties of the function that are not easily detected in the Boolean domain.

Function Complexity

As described in the previous section, the estimate $C(f)$ of a function's complexity may be computed using its spectrum. In the autocorrelation domain, the first-order autocorrelation coefficients may be used to compute $C(f)$. Each first-order coefficient b_u, $\|u\| = 1$, provides the number of 1-adjacencies with respect to the corresponding input variable. The number of 0-adjacencies is easily derived from the number of 1's in the function and the number of 1-adjacencies. Thus

$$\begin{aligned} C(f) &= \sum_{\|u\|=1} (b_u + (2^n - b_u) - 2(b_0 - b_u)) \\ &= n2^n - 2(nb_0 - \sum_{\|u\|=1} b_u). \end{aligned}$$

Application to Circuit Synthesis

The same linearization procedure [48] that may be performed on a function's spectrum may be performed on **B** to move the largest coefficients to the lowest order positions.

The realization of $f(X)$ would have the identical form to that depicted in Figure 2.3.

Function Classification

The autocorrelation function also may be used to perform function classification. The procedure is identical to the spectral classification procedure.

2.4 Summary

This chapter provides background material and definitions for an understanding of the chapters that follow, which describe applications of the autocorrelation of switching functions to logic optimization and to multi-level logic synthesis.

Chapter 3 describes its use in PLA optimization to obtain near-optimal input variable pairings for decoded PLAs. Performing this optimization almost always reduces, but never increases the size of the core of a PLA. The optimization technique also has particular relevance in implementing switching circuits with PLD-based FPGAs such as those produced by Altera [6] and AMD.

The work described in Chapters 5, 6, and 7, is applicable in synthesizing switching circuits to be implemented using random logic, and sea-of-gates FPGAs, particularly the Motorola MPA1000 [71] and Xilinx XC8100 [99] families of fine-grained FPGAs.

Chapter 6 examines whether linearizing a system of functions to maximize $C(f)$ does lead to simpler realizations. It finds that two-level realizations of the type

targeted to PLAs tend to become simpler; however, multi-level realizations suitable for FPGA implementation are not simplified by linearization.

The chapter also incorporates linearization into a synthesis method called two-place decomposition, thereby combining two synthesis methods to produce realizations for systems of functions.

In Chapter 7 the knowledge of the number of spectral-equivalent classes for functions of ≤ 5 variables is exploited to partition a large function into a number of smaller ones that are suitable for synthesis by the two-place decomposition method.

Chapter 3

PLA Optimization

The goals of PLA optimization are to minimize the area required by a PLA and to minimize the propagation delay through a PLA. Because of the regular structure of a PLA, the area required is proportional to the number of product terms in the array.

As seen in Chapter 2, Figure 2.1a depicts the core of an example PLA. In general, the area of the core of a PLA is

$$(L_{in} + L_{out}) \times P,$$

where L_{in} is the number of core input lines, L_{out} is the number of output lines, and P is the number of product terms.

The PLA core in Figure 2.1a is that of a *standard* PLA. In such a PLA there are two input lines generated to the core of the PLA for each function input variable: one for the variable's value and the other for the variable's complement. For a standard PLA implementing an n-input m-output function, $L_{in} = 2n$ and $L_{out} = m$.

Again, due to a PLA's regular structure, the delay is also proportional to the number of product terms. Thus *Boolean minimization* is of great importance for optimizing PLAs.

Efficient Boolean minimization methods [13, 23, 36, 82] have been developed for determining a canonical form that is minimal or near-minimal and targeted to PLA implementation. However, there are other logic optimizations that can be performed. Among these are *input and output encoding* [79, 80], *output phase optimization* [79, 82, 84], and *input decoding* [18, 79, 81, 82, 84].

Often there is the possibility of specifying the encoding of the inputs (or outputs) of a PLA so that the size of the core of the PLA is reduced. One example in which this is possible is in the design of a finite state machine, where the designer is almost always free to choose appropriate assignments to the state variables in order to optimize the area of the PLA.

Output phase optimization is the process of choosing, independently for each output, whether the core of the PLA generates the output value or its complement. The correct values of the outputs are generated by the output buffers of the PLA by having the appropriate buffers generate complemented values. Often the size of the PLA can be reduced by making the appropriate choices.

For the input decoding optimization, the input variables are partitioned into disjoint subsets and the variables in each subset serve as the inputs to a decoder [18, 81, 82, 84]. The outputs of the decoders, instead of the input signals and their complements, are used in the core of the PLA. This type of a PLA is called a *decoded* PLA.

In the general case the input variables $X = \{x_1, x_2, \ldots, x_n\}$ may be partitioned into d disjoint subsets $X_i, i = 1, 2, \ldots d$ where $1 \leq |X_i| \leq n$. Each decoder has $|X_i|$ inputs and $2^{|X_i|}$ outputs that serve as inputs to the PLA core. Each subset X_i is viewed as a single $2^{|X_i|}$-valued variable and the optimization of the decoded PLA is done via *multiple-valued minimization* [79].

The assignment of input variables to decoders must be such that the area of the resulting core of the PLA is minimal. In the general case, this task is extremely

difficult and time consuming. Thus to make the problem manageable, the decoders are limited in size. The approach used here, as in [81, 82, 84], is to set a limit k and to partition the PLA inputs into groups of size k with each group assigned to a decoder. Unless the number of PLA inputs is a multiple of k, a single decoder with less than k inputs is used for the final group.

For a standard PLA, which actually uses one-bit decoders, k PLA inputs yield $2k$ input lines to the core. For a decoded PLA, assigning those k inputs to a single decoder produces 2^k input lines to the core. For $k = 2$, the number of core inputs is the same as for a standard PLA and hence the difference in the area of the core is determined by the difference in the number of product terms. However, for larger values of k, the number of core input lines grows exponentially for the decoder case but linearly for the standard PLA. Thus the use of small decoders, say $k \leq 3$, is likely to be of most benefit since a considerable decrease in the number of product terms is required to offset the increase in the number of input lines caused by larger decoders.

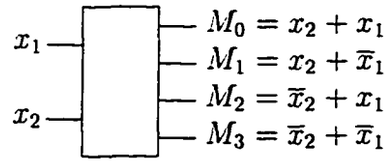
PLA optimization using input decoders is the thrust of this chapter, in which the cases of $k = 2$ and $k = 3$ are considered. This chapter describes a technique for assigning pairs of input variables to two-bit decoders, where variable pair selection is based on the total autocorrelation of a system of Boolean functions. Some results on the use of autocorrelation coefficients for the selection of variables to be assigned to three-bit decoders are also presented.

3.1 Introduction

Recall that any function may be represented as a product of maxterms. In the two-variable case

$$f(x_1, x_2) = (c_0 + x_2 + x_1)(c_1 + x_2 + \bar{x}_1)(c_2 + \bar{x}_2 + x_1)(c_3 + \bar{x}_2 + \bar{x}_1).$$

Figure 3.1: Two-bit Decoder



where $c_i, i = 0, 1, 2, 3$, is a constant 0 or 1. To realize a function represented by (c_0, c_1, c_2, c_3), the maxterms for which $c_i = 0$ are ANDed together.

Each of the 2^k outputs of a k-bit decoder is one of the 2^k possible maxterms of a k-variable function. Figure 3.1 depicts a two-bit decoder which generates the four maxterms of two variables.

In a decoded PLA, the maxterms that are generated by all of the decoders are available on every product line. Therefore each product term of a minimal sum-of-products expression for the system of functions may be implemented by selecting all of the appropriate maxterms on a single product line.

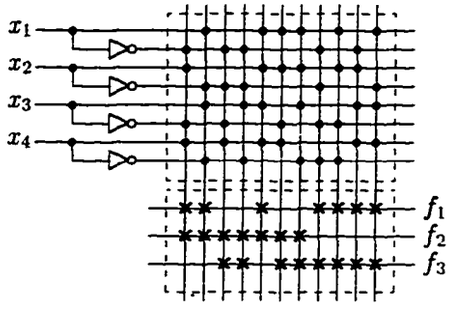
For example, if x_i and x_j are assigned to the same decoder, then literal x_i is realized by $x_i = (x_i + x_j)(x_i + \overline{x}_j)$. Every literal in a product term is realized by the product of the appropriate maxterms, and the product term is realized by forming the product of these products of maxterms, all on a single product line. Hence a system of functions may be implemented with a decoded PLA that requires no more product lines than that required in a standard PLA implementation.

This, coupled with the fact that a decoded PLA using two-bit decoders has the same number of core input lines as a standard PLA, means that the core of a decoded PLA using two-bit decoders can be no larger than the core of a standard PLA implementing the same system of switching functions. In fact, a decoded PLA using two-bit decoders is almost always smaller than a standard PLA. This analysis ignores

Table 3.1: Example Function

x_4	x_3	x_2	x_1	f_1	f_2	f_3
0	0	0	0	1	0	1
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	1	0	1
0	1	0	0	0	1	1
0	1	0	1	1	1	0
0	1	1	0	0	0	0
0	1	1	1	0	1	1
1	0	0	0	0	1	1
1	0	0	1	0	0	0
1	0	1	0	1	1	0
1	0	1	1	0	1	1
1	1	0	0	0	0	0
1	1	0	1	1	0	1
1	1	1	0	1	0	1
1	1	1	1	1	1	0

Figure 3.2: Standard PLA Realization



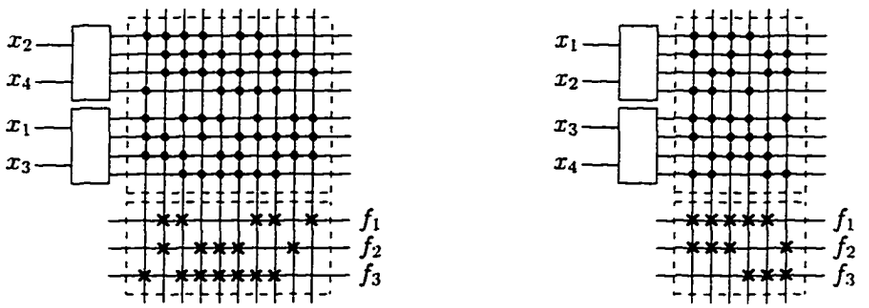
the sizes of the decoders themselves.

The size of the core of a decoded PLA is significantly affected by the choice of variables assigned to each decoder. For example, Table 3.1 shows the truth table for a system of functions and Figure 3.2 depicts its standard PLA realization. Figures 3.3(a) and 3.3(b) show decoded PLA realizations for the same system with an arbitrary pair assignment and an optimal pair assignment respectively. As Figure 3.3 shows, the area required by the PLA with an optimal pair assignment is significantly less than the PLA with the arbitrary assignment.

This chapter describes an algorithm that determines a near-optimal input variable pairing for a system of Boolean functions from information contained in the system's total autocorrelation. The algorithm produces excellent results in comparison to other approaches to this problem. In particular, the approach is effective for PLAs with large numbers of inputs and outputs.

The PLA pairing problem is a special case of the more general problem of selecting variable groupings for multi-bit decoder PLAs. Application of the autocorrelation to

Figure 3.3: Decoded PLA Realizations of Example Function



this more general problem is also discussed briefly. In particular, results are provided for the assignment of variables to three-bit decoders and an assignment scheme in which three variables are assigned to one three-bit decoder and the remaining variables are assigned to two-bit decoders.

3.2 The PLA Pairing Problem

The PLA variable pairing problem may be stated as follows: given a system of Boolean functions, determine an optimal assignment of the input variables to the two-bit decoders for a decoded PLA. An optimal pairing of the input variables is an assignment that minimizes the area of the core of the final PLA. Note that the area requirements of the decoders are not considered here.

There are $\frac{n!}{2^m \cdot m!}$ possible pair assignments for n variables, where $m = \lfloor \frac{n}{2} \rfloor$. An exhaustive search procedure may be employed to find an optimal solution; for every pairing, the system of functions must be minimized to determine the pairing's effectiveness. For small values of n, say $n \leq 12$, an optimal assignment can be found using this method. However, as n grows, the computational requirements make this

approach infeasible. Therefore, algorithms that employ heuristics are needed to arrive at near-optimal solutions relatively quickly.

Some heuristic algorithms that obtain "good" solutions relatively quickly have been developed [18, 82, 84] and have been shown to be quite effective at reducing the area required by a PLA. Each of the algorithms begins by obtaining a near minimal two-level canonical form for the system of functions from which an *assignment graph* is constructed. This is a complete weighted graph of n nodes. Each node represents an input variable x_i and the weight of each edge (i, j) represents an estimate of the complexity of the minimized function if variables x_i and x_j are paired. This estimate is the number of unique product terms that remain in the near-minimal canonical expression after the variables x_i and x_j and their complements are removed.

Sasao [82, 84] addresses the problem of assigning variables to two-bit decoders in a decoded PLA. This is accomplished by assigning to a decoder the variables x_i and x_j for each edge (i, j) in a minimum-weight maximum-cardinality matching of the assignment graph. The algorithm achieves an average 20% to 30% area reduction over standard PLAs.

Chen and Muroga [18] consider the more general grouping problem where one may use multi-bit decoders to reduce further the area required by a PLA. First, they order the input variables by finding a minimum-weight Hamiltonian path in the assignment graph. Then a minimum cost partitioning of the variables is found using dynamic programming and one decoder is assigned to each partition. The cost function used by this algorithm takes into consideration the area overhead of the decoders and of any additional input lines that are introduced by them.

A major drawback to the above approaches is that an initial minimal or near-minimal two-level canonical expression for the system of functions is required to build the assignment graph. Also, the processing of the assignment graph is computationally expensive.

The following section describes a heuristic algorithm which is based on the total autocorrelation of a system of functions. It does not need a near-minimal canonical expression from which to begin nor does it perform any optimization of a structure such as the assignment graph. Variable selection is done directly from the second-order autocorrelation coefficients.

3.3 Application of the Autocorrelation

Each second-order coefficient, b_u, $\|u\| = 2$, for a given function provides a measure of the similarity between the values the function takes when the pair of variables x_i and x_j, $u_i = u_j = 1$, are assigned values that are Hamming distance two apart. The minimal sum-of-products expression needed to specify function values that are at distance two from each other requires two product terms (to specify the exclusive-or operation). In a standard PLA realization, two product terms are necessary.

In a decoded PLA realization, only one product term is needed. Further, any subset of the four possible assignments to variables x_i and x_j can be specified in only one product term in a decoded PLA. To see why only one product term is required consider the following.

One can view the specification of any subset of the four assignments to x_i and x_j as a 2-variable function which takes the value 1 at each of those assignments. Recall that the decoder to which x_i and x_j are inputs generates all the maxterms of x_i and x_j. Recall also that any switching function may be expressed as a product of its maxterms. Thus the function defining the subset of assignments to x_i and x_j may be implemented with a single product term.

In choosing a pairing, the variables that should be paired are the ones that take most advantage of this fact. Thus the variables that should be paired are the ones

Algorithm 3.1: Autocorrelation Pairing

Compute the second-order total autocorrelation coefficients b_u, $\|u\| = 2$
While a coefficient remains
{
Consider all remaining b_u in order $u = 3, 5, 6, \ldots$
determine a b_v with maximum value selecting the last one encountered in the case of a tie
Determine i and j where $v_i = v_j = 1$
Pair the variables x_i and x_j
Remove all b_u such that $u_i = 1$ or $u_j = 1$
}

whose corresponding second-order autocorrelation coefficients have largest value. This is the basis of the pairing algorithm.

3.3.1 The Autocorrelation Pairing Algorithm

Algorithm 3.1 provides the autocorrelation pairing algorithm. For an example of its operation, consider the function in Table 3.1. Table 3.2 gives the autocorrelation coefficients $\mathbf{B}_i(u)$ of f_i as well as the total autocorrelation coefficients $\mathbf{B}(u)$. This is the same table given in Table 2.6. For clarity it is repeated here with the second order coefficients highlighted in underlined bold type.

From the table, the largest-value second-order total autocorrelation coefficient is $b_3 = 16$, which corresponds to variables x_2 and x_1 since $u_2 = 1$ and $u_1 = 1$. Thus x_2 and x_1 are chosen as a pair to be assigned to a decoder. This eliminates all second-order coefficients $b_v, v \in \{5, 6, 9, 10\}$ since for each, $v_1 = 1$ or $v_2 = 1$. Then the largest (and only) remaining second-order coefficient is $b_{12} = 10$ which corresponds to variables x_4 and x_3. Therefore these two input variables are also paired. The

Table 3.2: Autocorrelation Coefficients for Example

v, u	f_1	f_2	f_3	R_1	R_2	R_3	B_1	B_2	B_3	B
0000	1	0	1	7	7	8	7	7	8	22
0001	0	0	0	-1	-1	0	2	4	0	6
0010	0	0	0	-1	-1	0	2	4	0	6
0011	1	0	1	-1	3	4	4	4	8	**16**
0100	0	1	1	-1	-1	0	2	2	4	8
0101	1	1	0	3	3	0	4	2	4	**10**
0110	0	0	0	-1	-1	0	2	0	4	**6**
0111	0	1	1	3	-1	4	2	2	4	8
1000	0	1	1	-1	-1	0	2	2	4	8
1001	0	0	0	-1	-1	0	2	0	4	**6**
1010	1	1	0	3	3	0	4	2	4	**10**
1011	0	1	1	3	-1	4	2	2	4	8
1100	0	0	0	3	-5	0	2	4	4	**10**
1101	1	0	1	-1	-1	0	4	4	4	12
1110	1	0	1	-1	-1	0	4	4	4	12
1111	1	1	0	3	-1	-4	4	6	4	14

pairing chosen for this particular example is $(x_2, x_1)(x_4, x_3)$ and is the optimum one (as verified by exhaustive search).

3.4 Benchmark Circuit Results

The autocorrelation pairing algorithm is implemented in the C programming language and is interfaced with the multiple-valued minimizers ESPRESSO [13] and MINI II [82]. Near-optimal pairings chosen by the autocorrelation pairing procedure for several benchmark functions are presented in this section and are compared with the results of other researchers. In addition, Tables 3.3 and 3.4 compare the results with optimal pairings obtained by exhaustive search. The only criterion used for evaluation of the algorithm is the number of product terms needed in the core of the decoded PLA. No consideration is made of the cost of the decoders.

Tables 3.3 and 3.4 provide the results for a set of arithmetic functions and a set of control functions respectively. The information provided in the tables for each

Table 3.3: Arithmetic Function Pairing Results
*- from 1500 randomly selected pairings

PLA	# pi/po	sop	Exhaustive max	Exhaustive avg	Exhaustive min	AC pair	Sasao	Chen & Muroga
add6	12/7	355	346	253.3	37	37	37	—
adr3	6/4	31	25	20.5	10	10	10	—
adr4	8/5	75	67	50.7	17	17	17	—
dist3	6/4	36	31	28.1	19	19	19	—
dist4	8/5	120	105	94.3	71	74	74	73
fl6	6/6	40	36	33.5	31	32	34	—
fl8	8/8	129	120	110.5	102	110	112	—
mdiv7	8/10	203	180	156.1	97	113	119	—
mlp2	4/4	7	7	6.3	6	6	6	—
mlp3	6/6	31	29	26.1	21	23	23	—
mlp4	8/8	124	115	104.6	89	88	89	—
root6	6/4	23	21	19.8	17	18	18	—
root7	7/4	34	33	29.5	25	25	26	27
root8	8/5	57	53	48.8	38	42	38	42
root9	9/5	85	82	73.9	60	61	62	64
sqr4	4/8	13	13	11.7	11	11	11	—
sqr6	6/12	48	46	42.0	40	41	41	—
sqr8	8/16	180	169	160.8	151	158	156	—
sn74181	14/8	575	569*	524.6*	300*	288	282	—

function is divided into columns as follows:

PLA function name

pi/po number of inputs and outputs respectively

sop number of product terms required in a standard PLA as determined by ESPRESSO

Exhaustive

maximum, average, and minimum number of product terms required in a decoded PLA over all possible pairings. The results were obtained using ESPRESSO.

Table 3.4: Control Function Pairing Results
*- symmetric function

PLA	# pi/po	sop	Exhaustive max	Exhaustive avg	Exhaustive min	AC pair	Sasao	Chen & Muroga
2of5*	5/1	10	5	5.0	5	5	5	—
postal	8/1	18	18	12.8	8	9	9	—
5xp1	7/10	64	61	54.5	46	49	47	46
9sym*	9/1	85	28	26.4	26	29	28	28
bw	5/28	22	22	22.0	22	22	22	—
con1	7/2	9	9	8.3	7	7	8	—
f2	4/4	8	6	6.0	6	8	8	—
f51m	8/8	76	66	58.5	48	50	52	50
misex1	8/7	12	12	11.7	10	11	10	—
rd53*	5/3	31	13	12.2	12	12	12	—
rd73*	7/3	127	38	37.8	37	37	37	37
rd84*	8/4	255	54	54.0	54	54	54	—
sao2	10/4	58	48	42.7	36	37	38	—
alu2	10/8	68	68	60.2	41	41	41	—
alu3	10/8	65	65	57.8	36	37	36	—
clip	9/5	118	112	92.9	40	43	40	—
dc2	8/7	39	39	35.9	31	33	32	—
dk17	10/11	18	18	18.0	18	18	18	18
risc	8/31	28	28	27.9	27	28	27	—
sqn	7/3	38	36	32.8	25	33	28	27
xor5*	5/1	16	4	4.0	4	4	4	—
z4	7/4	59	49	38.7	16	16	16	16

AC pair

number of product terms required in a decoded PLA using the pairing selected by the autocorrelation pairing algorithm. For comparison with Sasao's algorithm, MINI II was used to perform the minimization.

Sasao number of product terms required in a decoded PLA using the pairing selected by Sasao's pairing algorithm [82]. Computer software obtained from Sasao [85] selected the pairings and MINI II performed the minimization.

Chen & Muroga

number of product terms required in a decoded PLA as reported in [18].

Both ESPRESSO [13] and MINI II [82] implement heuristic-based algorithms and hence produce results that may not be strictly minimal. Also since the heuristics used in ESPRESSO differ from those used by MINI II, there may be variations in the results produced by each for the same problem. This is exemplified in Table 3.3 for *mlp4*, where MINI II produces a better result than the best result found by exhaustive search using ESPRESSO.

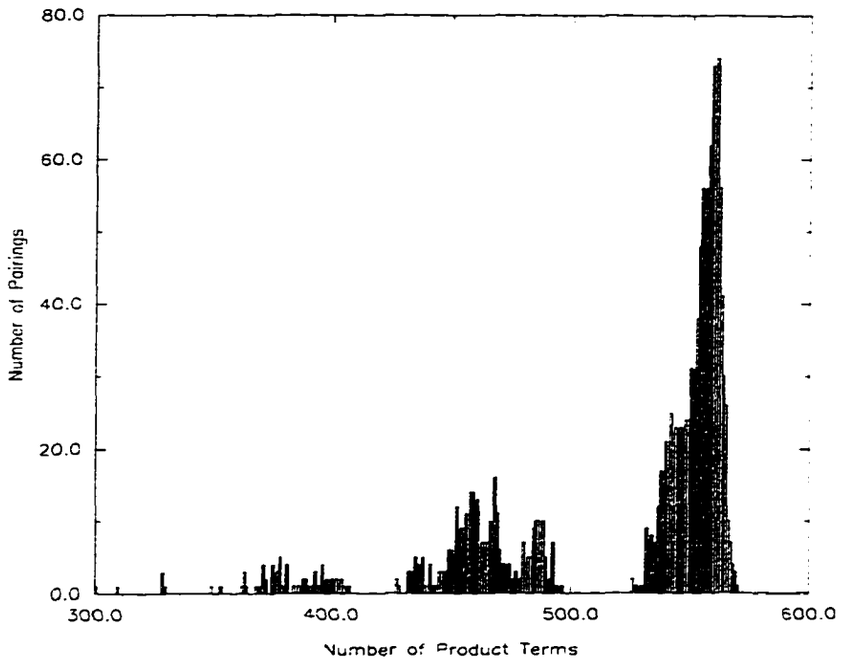
Also, factors such as the order in which cubes are specified or the order in which pairs are specified affects the quality of the results produced by each of the programs. For example, ESPRESSO produces varying numbers of product terms in the exhaustive search data for symmetric functions *9sym*, *rd53*, and *rd73* even though all possible pairings of a symmetric function requires exactly the same number of product terms in a decoded PLA. Therefore a variation of one or two product terms among the reported results of the different researchers for a particular function is not necessarily significant.

For the arithmetic function *sn74181*, an exhaustive search is not feasible due to the large number of possible pairings. Therefore 1,500 random pairings, chosen out of the possible 135,135, are used to obtain an estimate of the number of product terms required for a decoded PLA realization. The histogram in Figure 3.4 provides the results.

Each bar in the histogram indicates the number of pairings that require a given number of product terms in a decoded PLA. The mean number of product terms required in a decoded PLA is computed by summing the number of product terms required by each pairing and dividing by the number of selected pairings. For these 1,500 randomly selected pairings, the mean number of product terms required in a decoded PLA is 524.

Figure 3.4 clearly shows that for the 1,500 selections, the vast majority of the results are far from minimal, with most lying above the mean, which is consequently

Figure 3.4: sn74181 Random Pairing Results



also far from the minimum. The result for the best pairing of the 1,500 is also significantly larger than the result for the pairing found by the autocorrelation pairing algorithm.

The data in Figure 3.4 is clustered and not a continuous distribution. This is likely due to partial symmetries in the functions and also the fact that the appropriate pairing of certain variables has a greater effect than the pairing of others.

Histograms of the exhaustive data of several of the functions listed in Tables 3.3 and 3.4 exhibit the same general structure as that given in Figure 3.4. Specifically there are very few optimal or near-optimal pairings and the huge majority of the pairings are far from minimal. This indicates that the autocorrelation pairing procedure finds a good result because the heuristic is a good one and not merely because there happens to be several from which to choose.

A more detailed analysis of the nature of these histograms is beyond the scope of this work but may prove profitable in lending more insight into how best to pair PLA inputs.

These experimental results indicate that the autocorrelation pairing procedure works quite well. It selects an optimal pairing for 20 of the 40 functions for which exhaustive search data are listed. These includes functions *9sym* and *f2* for which all pairings require the same number of product terms but for which MINI II generates results that are larger than the largest found by exhaustive search. For all but four functions, *f18*, *mdiv7*, *sqr8*, and *sqn*, the pairing selected is within five product terms of the optimum.

The autocorrelation pairing results also compare favourably with those of Sasao. Of the 19 functions listed in Table 3.3 the autocorrelation procedure produces better results in 5 cases and worse results in 3 cases. Of the 22 functions in Table 3.4 the autocorrelation procedure produces better results in 3 cases and worse results in 7 cases. In total, for the 41 functions listed, the autocorrelation procedure produces better results in 7 cases and worse results in 10 cases with a variation of more than two product terms for five functions, most significantly for *sn74181*, *sqn*, and *mdiv7*.

For the arithmetic functions, the autocorrelation pairing procedure performs quite well since these functions tend to exhibit significant XOR structure. The control functions however have less XOR structure and hence the autocorrelation procedure does not perform as well as Sasao's algorithm. However, some of these results are improved with enhancements to the autocorrelation paring algorithm as discussed in the next section.

The comparison with Chen and Muroga's algorithm is similar. The autocorrelation procedure produces results that are better than those of Chen and Muroga for 2 of the 11 reported results, worse for three, and the same for the remaining 6 results (including *9sym*). The variation between the results is at most three product terms

Table 3.5: Large Function Pairing Results

PLA	# pi/po	sop	AC Pair prod terms	AC Pair select time	Sasao prod terms	Sasao select time
apex5	117/88	1088	689	27.82	651	19573.74
intb	15/7	631	350	7.05	295	6.10
misex3	14/14	662	478	5.68	456	6.74
misg	56/23	69	53	1.0	48	6.07
mish	94/43	82	59	1.4	55	344.40
misj	35/14	35	23	0.12	23	12.45
sn74181	14/8	575	288	7.63	283	1.31
vg2	25/8	110	88	0.31	88	0.31
x2dn	82/56	104	74	1.2	73	255.15
x7dn	66/15	538	272	16.4	278	20.70
xparc	41/73	254	211	2.5	213	4.11

except for *sqn* where Chen and Muroga's result is better by 6 product terms or 18%. Again this is improved with enhancements described in the following sections.

Table 3.5 provides results for a set of large functions. It includes for each function, as in Tables 3.3 and 3.4, the name of the function, the number of inputs and outputs, and the number of product terms required in a standard PLA. Also the table provides for each of the autocorrelation pairing algorithm and Sasao's pairing algorithm, the number of product terms needed for a decoded PLA using the selected pairing as well as the execution time used to do the pair selection.

For each of these functions the first and second-order autocorrelation coefficients are computed directly from the list of cubes defining the function. The method for computing the required autocorrelation coefficients directly from a cube list has two steps. First the disjoint sharp operation [27] is used to convert the cube list to a disjoint cover (one where no two cubes intersect). Then the cubes are analyzed singly and in pairs to determine the contribution they make to the first and second order autocorrelation coefficients. Note that both the first-order and second-order

autocorrelation coefficients are required for the enhancements discussed below.

The complexity of this process depends upon the number of cubes in the disjoint cover and is thus highly function dependent. It is exponential in the number of variables in the worst case, but for most "practical" functions runs in reasonable time.

The pairing selection time for the autocorrelation pairing algorithm includes the time required to compute the first and second-order autocorrelation coefficients. For Sasao's algorithm, software provided by Sasao [85] was used and the reported times are for the selection of the pairing only and do not include the time needed to generate the initial near-minimal two-level canonical form that is required. Both algorithms were performed on a Sun 4/370 computer system and the timings were obtained using the Unix system timing facilities. MINI II [82] was used to do the minimization for the pairings selected by both algorithms.

For most of these large functions Sasao's algorithm selects better pairings with the most significant differences for the functions *apex5*, *intb*, and *misex3*. As shown in the next section, these results are improved with some enhancements to the basic autocorrelation pairing algorithm. Of more significance are the timings. The timings show that the autocorrelation approach can be significantly more efficient than Sasao's approach.

The autocorrelation procedure is considerably faster for the larger functions. This is due to the fact that the assignment graph that Sasao's algorithm uses becomes quite large. For an n-input function the graph has n nodes and $O(n^2)$ edges and for each edge a weight must be generated. Generation of the edge weights is a non-trivial task. Also finding the minimum-weight maximum-cardinality matching becomes correspondingly more time consuming as the number of inputs grows.

The autocorrelation pairing procedure on the other hand does not use such a structure and computes the first and second-order autocorrelation coefficients directly

from the cubes and its time complexity depends on the number of cubes in the disjoint cover. Thus if the number of cubes is small relative to n, as for functions *apex5*, *mish*, and *x2dn*, the autocorrelation procedure requires less time to select a pairing. However, if the number of cubes is large relative to n, then the autocorrelation pairing procedure requires more time than Sasao's algorithm as seen for functions like *sn74181* and *intb*.

3.5 Remarks

The autocorrelation algorithm for near-optimal input variable assignment for two-bit decoded PLAs does not require an initial near-minimal canonical expression. Also, it does not perform any optimization of a structure such as the assignment graph. Variable selection is done directly from the second-order autocorrelation coefficients.

Since fewer steps are involved and the computation for an initial near-minimal canonical form is not necessary, the algorithm is potentially more efficient than those of Sasao [82, 84], as can be seen in the timings provided in Table 3.5. Since Chen and Muroga [18] also use an assignment graph and in addition determine a Hamilton path for it, their algorithm requires more time than Sasao's algorithm. Hence the autocorrelation pairing algorithm can be more efficient than Chen and Muroga's algorithm as well.

The results of the autocorrelation pairing procedure compare favourably with those of Sasao, and Chen and Muroga. In only a few cases it produced notably poorer results than those of Sasao or Chen and Muroga and several of those are improved with the following enhancements.

3.5.1 Threshold Heuristic Enhancement

In the above algorithm, variables are paired if the corresponding second-order autocorrelation coefficient has a large value. However, if a first-order coefficient for either of the variables is *significantly larger* than the second-order coefficient, then the variables should not be paired. The reason is that a much larger first-order coefficient tends to indicate that the function is somewhat independent of the corresponding variable and pairing it with another variable provides little benefit.

Indeed, if the function is completely independent of variable x_i, then x_i is not included in any minimal switching expression for the function. Hence, for any pair (x_i, x_j), the only maxterms that are used in a decoded PLA are those whose product is x_j or $\overline{x}_j$. This has the effect of using a one-bit decoder with x_j for its input, as in a standard PLA, and therefore has no value for product term reduction.

A coefficient can be considered *significantly larger* than another one if the difference between the two is greater than or equal to a given threshold value. By studying the examples cited above, a suitable threshold is found to be the value $\frac{2^n}{2^{m+1}}$, where n is the number of inputs and m is the number of outputs for the system of Boolean functions.

The pairing algorithm is modified here to choose a second-order coefficient only if it is significantly larger than the relevant first-order coefficients. If, at some point, no second-order coefficient meets this criterion, then any remaining variables are paired using the original selection scheme.

Table 3.6 gives the results for those functions that are affected by this change. As can be seen the results for all but two of these functions, namely *apex5* and *misg*, are improved.

Both of these functions have a relatively large number of outputs. For any single output, there may not be a large difference between autocorrelation coefficient values.

Table 3.6: Functions Affected by Threshold Heuristic

PLA	sop	Exhaustive		AC Pair	
		max	min	old	new
mdiv7	203	180	97	113	100
root8	57	53	38	42	38
root9	85	82	60	61	60
sqn	38	36	25	33	29
apex5	1088	—	—	689	759
misex3	662	—	—	478	456
misg	69	—	—	53	63

However, since the total autocorrelation coefficients accumulate the contributions of each output a large variance may result between total autocorrelation coefficients. Thus if some first-order autocorrelation coefficient is consistently larger than any of the second-order coefficients for all outputs, then the first-order total autocorrelation coefficient may be significantly larger than the second-order coefficients. This large difference may cause the threshold heuristic to reject a good pairing.

3.5.2 Minimal-Variance Heuristic Enhancement

In this enhancement, the variables that are chosen are those for which the corresponding first and second-order autocorrelation coefficients have minimal variance among their values.

For each possible pair of variables a weighted variance value is computed for the three corresponding autocorrelation coefficients. The weights are given such that the most desirable variables to be paired are those for which the corresponding first and second order coefficients are all equal. The least desirable variables are those for which all the corresponding first and second order coefficients have different values.

The rationale behind this heuristic is as follows. As the value of a first-order coefficient tends toward a maximum, the more independent of the corresponding variable

Table 3.7: Functions Affected by Minimum Variance Heuristic

		Exhaustive		AC Pair	
PLA	sop	max	min	old	new
clip	118	112	40	43	40
mdiv7	203	180	97	113	173
sn74181	575	—	—	288	281
sqn	38	36	25	33	25
intb	631	—	—	350	295
misex3	662	—	—	478	490
x7dn	538	—	—	272	294

is the function and therefore the variable should not be involved in a pairing. A larger valued second-order coefficient indicates that the function has similar columns at distance two with respect to the corresponding input variables and therefore those variables should be paired. If more than 2 columns are similar with respect to a pair of variables, the differences in value among the corresponding first and second-order coefficients are small. Indeed, as shown by Theorem 6.1, if three columns are equal, then the corresponding first and second-order coefficients have the same value.

Table 3.7 reports the significant results. As the table shows, this approach improves several of the pairing results. However when it makes a result worse, the effect is quite severe. An explanation for this is that while the variance between the corresponding first and second-order coefficients may be minimal for a particular pair of variables, that does not guarantee that more than two columns are similar. Furthermore, since the total autocorrelation is computed as the sum of the autocorrelations of the individual outputs of a multiple-output function, major differences in value between corresponding first and second-order coefficients for one output could cancel the differences for another output.

3.5.3 Other Attempts at Improvement

Another attempt to improve the results uses a data structure similar to the previously mentioned assignment graph. In this structure the weight on an edge of the graph is the second-order autocorrelation coefficient value corresponding to the variables joined by the edge. A pairing is obtained by finding a maximum-weight maximum-cardinality matching of the graph. The effect of this procedure is to pair the variables corresponding to the b_u, $\|u\| = 2$ such that $\sum_{\|u\|=2} b_u$, all b_u disjoint, is maximum.

The results obtained are not significantly different from the previous results. Some pair assignments are slightly better while others are slightly worse. The majority of the results are the same. The overall conclusion is that the use of this structure does not lead to significantly different results. Furthermore, this procedure requires significantly more computation time due to the search for the matching of the graph. Therefore this approach is inferior to the previously described autocorrelation pairing procedures.

3.5.4 Extension for General Grouping

In a more general grouping, multi-bit rather than two-bit decoders may be used. These decoders require much more area than the two-bit decoders since not only are they themselves larger, but they also introduce more input lines into the core of the PLA. For example a four-bit decoder introduces sixteen input lines into a PLA. This is double the number required by the four one-bit decoders used in a standard PLA realization. Hence, this increase in area may override any savings in area due to a reduction in the number of product terms in the PLA. Currently only the area required for the core of the PLA for a particular grouping is used to evaluate its effectiveness.

Here, the application of the above algorithm to finding triple assignments for the

input variables is examined. A triple assignment is one in which as many input variables as possible are assigned to three-bit decoders and the remaining one or two variables are assigned to a one-bit or two-bit decoder respectively. Another type of grouping that is examined is one in which only one three-bit decoder is used and a pairing assignment is found for the remaining variables. This grouping is called a *1-triple pairing.*

Near-optimal groupings for these structures are selected with a technique similar to the one used for pairing selection. For a triple assignment, the algorithm is identical to the pairing algorithm except that the variables corresponding to the largest-value third-order autocorrelation coefficient are assigned to a three-bit decoder.

For a 1-triple pairing, the first operation performed is to assign, to a three-bit decoder, the variables corresponding to the largest third-order coefficient. Then a pairing assignment is found for the remaining variables using the pairing algorithm.

The results for the near-optimal groupings found for a set of benchmark functions are presented in Table 3.8. These results are compared with optimal groupings found by exhaustive search. The values in the table are the number of product terms required in the core of the PLA.

Table 3.9 provides, for each benchmark function, the area required by the core of a standard PLA and, for each grouping structure, the ratio, specified as a percentage, of the area required by the core of the decoded PLA to that of the standard PLA. As can be seen in this table a significant reduction in area may be achieved over that obtained by a pair assignment.

The tables also show that for the triple and 1-triple pair structures the autocorrelation procedure does not perform as well as for the pair structure. They also indicate that, although there may be a reduction in the number of product terms, the area required by the core of the PLA may increase.

3.6 Conclusion

This chapter examines the use of non-traditional logic design techniques for the optimization of PLAs. In particular, a new method for pairing variables in a two-bit decoder PLA is developed. Also it provides results of an investigation into extending these techniques to PLAs with three-bit decoders. Multiple-valued minimization is used to optimize the PLA once the input groupings are selected.

Since the autocorrelation algorithm is a heuristic one, there are instances where the results it obtains are inferior to those obtained by other methods. This can be seen in the above tabulations. However, the autocorrelation pairing algorithm yields results comparable to earlier approaches. The key advantages are the relative simplicity of the method and its efficiency. These allow the user to try any or all of the variants of the algorithm and use the best results that are obtained.

The techniques described in this chapter are discussed specifically with respect to PLAs. However, these techniques are also valuable when applied to FPGA technology. specifically PLD-based FPGAs such as those produced by Altera [6] and AMD [4]. PLDs are PLA-like structures where an AND array provides product terms that may be routed to OR cells to provide two-level implementations of functions. the values of which can be used as inputs to other AND arrays on the chip. Thus every level in a multi-level switching circuit essentially consists of a PLA.

For this type of FPGA, a minimal switching expression for a system of functions is one for which a minimum number of AND arrays is required. The AND arrays are of a maximum size. For example, Altera's EPM5128 [6] FPGA can implement up to 35 product terms in a single AND array. Thus, any system of functions that requires more product terms than are available in a single AND array needs to be partitioned to fit into two or more AND arrays.

If the same system of functions is optimized through input decoding, the number of

product terms may be reduced to allow it to be implemented using fewer of the FPGA logic blocks. This then allows more functions to be implemented on an individual FPGA chip which leads to a simpler and less costly overall system implementation. The total cost of implementing the decoded system of functions must of course include the cost of the decoders, which would also be realized using FPGA cells.

A possible new FPGA architecture is one which contains both PLDs and programmable two-bit decoder cells. The decoder cells would provide the decoded inputs and the PLDs would implement the logic as in an existing PLD-based FPGA, thus enabling the generation of the required maxterms without using the PLD resources, thereby making more efficient use of the chip.

Areas for future work include investigating ways to improve the algorithm for the larger grouping structures. In addition, the algorithm ideally should automatically select a grouping that gives the maximum reduction in area. This includes grouping structures that are not limited to those mentioned above. Also the area calculations should include the area required by the decoders themselves, especially when larger grouping structures are being considered.

Table 3.8: Grouping Product Term Comparison
*- symmetric function
†- from 2000 randomly selected groupings

PLA	# pi/po	sop	pair max	pair min	pair AC	triple max	triple min	triple AC	1-triple pair max	1-triple pair min	1-triple pair AC
add6	12/7	355	346	37	37	311†	45†	65	338†	66†	47
adr2	4/3	11	9	5	5	6	5	5	6	5	5
adr3	6/4	31	24	10	10	21	9	11	21	10	11
adr4	8/5	75	67	17	17	54	16	21	61	17	21
adr5	10/6	167	148	26	26	134	32	41	145†	33†	33
dist3	6/4	36	31	19	19	23	17	17	26	18	20
dist4	8/5	120	105	72	74	86	58	58	97	66	65
fl6	6/6	40	36	31	32	28	24	26	32	26	27
fl8	8/8	129	122	99	109	101	81	85	112	87	87
mlp2	4/4	7	7	6	6	6	5	5	6	5	5
mlp3	6/6	31	28	21	25	24	18	21	27	19	24
mlp4	8/8	124	114	90	88	101	71	79	111	77	89
root6	6/4	23	21	17	17	19	11	13	21	13	16
root7	7/4	34	33	25	25	29	17	18	29	19	19
root8	8/5	57	53	39	40	48	27	31	53	29	33
root9	9/5	85	81	60	62	69	41	43	78	48	50
sqr3	3/6	7	7	6	6	5	5	5	5	5	5
sqr4	4/8	13	13	11	11	11	9	9	11	9	9
sqr5	5/10	26	25	21	21	22	17	17	23	17	17
sqr6	6/12	48	45	40	41	39	31	32	41	33	34
sqr8	8/16	180	169	149	153	146	118	118	163	126	136
sn74181	14/8	575	569†	300†	288	550†	262†	265	—	—	250
2of5*	5/1	10	5	5	5	3	3	3	3	3	3
postal	8/1	18	18	8	9	12	5	6	15	7	7
5xp1	7/10	64	58	46	49	51	31	31	54	34	43
9sym*	9/1	85	28	26	26	10	10	10	18	17	17
bw	5/28	22	22	21	22	22	21	21	22	21	21
con1	7/2	9	9	7	7	9	6	7	9	6	8
f2	4/4	8	6	6	6	7	7	7	7	7	7
f51m	8/8	76	63	47	50	55	31	32	62	35	47
misex1	8/7	12	12	10	11	12	10	12	12	10	12
rd53*	5/3	31	13	12	12	7	7	7	7	7	7
rd73*	7/3	127	38	37	37	21	20	20	23	21	21
rd84*	8/4	255	55	54	54	30	27	27	48	43	43
sao2	10/4	58	48	35	37	44	28	31	48†	33†	37
alu1	12/8	19	—	—	19	—	—	18	—	—	19
alu2	10/8	68	67	41	41	66	32	34	67†	35†	33
alu3	10/8	65	65	37	37	63	28	29	65†	34†	29
apla	10/12	25	25	25	25	25	24	25	—	—	25
clip	9/5	118	112	40	43	89	35	44	108	37	48
dc1	4/7	9	9	8	9	9	7	7	9	7	7
dc2	8/7	39	39	31	32	39	24	24	39	26	26
dk17	10/11	18	18	18	18	18	17	18	—	—	18
dk27	9/9	10	10	10	10	10	10	10	10	10	10
risc	8/31	28	28	27	28	28	27	28	28	27	28
sqn	7/3	38	36	25	31	31	21	23	34	20	26
wim	4/7	9	9	8	9	8	7	7	8	7	7
xor5*	5/1	16	4	4	4	2	2	2	2	2	2
z4	7/4	59	48	16	16	44	11	20	44	10	20

Table 3.9: Grouping Area Percentage Comparison

PLA	# pi/po	sop	pair			triple			1-triple pair		
			max	min	AC	max	min	AC	max	min	AC
add6	12/7	11005	97.5	10.4	10.4	110.2	15.9	23.0	101.4	19.8	14.1
adr2	4/3	121	81.8	45.5	45.5	64.5	53.7	53.7	64.5	53.7	53.7
adr3	6/4	496	77.4	32.3	32.3	84.7	36.3	44.4	76.2	36.3	39.9
adr4	8/5	1575	89.3	22.7	22.7	85.7	25.4	33.3	89.1	24.8	30.7
adr5	10/6	4342	88.6	15.6	15.6	98.8	23.6	30.2	93.5	21.3	21.3
dist3	6/4	576	86.1	52.8	52.8	79.9	59.0	59.0	81.2	56.2	62.5
dist4	8/5	2520	87.5	60.0	61.7	85.3	57.5	57.5	88.5	60.2	59.3
fl6	6/6	720	90.0	77.5	80.0	85.6	73.3	79.4	88.9	72.2	75.0
fl8	8/8	3096	94.6	76.7	84.5	91.3	73.3	76.9	94.1	73.1	73.1
mlp2	4/4	84	100.0	85.7	85.7	100.0	83.3	83.3	100.0	83.3	83.3
mlp3	6/6	558	90.3	67.7	80.6	94.6	71.0	82.8	96.8	68.1	86.0
mlp4	8/8	2976	91.9	72.6	71.0	95.0	66.8	74.3	97.0	67.3	77.8
root6	6/4	368	91.3	73.9	73.9	103.3	59.8	70.7	102.7	63.6	78.3
root7	7/4	612	97.1	73.5	73.5	104.2	61.1	64.7	94.8	62.1	62.1
root8	8/5	1197	93.0	68.4	70.2	100.3	56.4	64.7	101.8	55.7	63.4
root9	9/5	1955	95.3	70.6	72.9	102.4	60.8	63.8	99.7	61.4	63.9
sqr3	3/6	84	100.0	85.7	85.7	83.3	83.3	83.3	83.3	83.3	83.3
sqr4	4/8	208	100.0	84.6	84.6	95.2	77.9	77.9	95.2	77.9	77.9
sqr5	5/10	520	96.2	80.8	80.8	93.1	71.9	71.9	97.3	71.9	71.9
sqr6	6/12	1152	93.8	83.3	85.4	94.8	75.3	77.8	92.5	74.5	76.7
sqr8	8/16	5760	93.9	82.8	85.0	91.2	73.8	73.8	96.2	74.4	80.3
sn74181	14/8	20700	99.0	52.2	50.1	116.9	55.7	56.3	—	—	45.9
2of5*	5/1	110	50.0	50.0	50.0	35.5	35.5	35.5	35.5	35.5	35.5
postal	8/1	306	100.0	44.4	50.0	82.4	34.3	41.2	93.1	43.5	43.5
5xp1	7/10	1536	90.6	71.9	76.6	93.0	56.5	56.5	91.4	57.6	72.8
9sym*	9/1	1615	32.9	30.6	30.6	15.5	15.5	15.5	23.4	22.1	22.1
bw	5/28	836	100.0	95.5	100.0	105.3	100.5	100.5	105.3	100.5	100.5
con1	7/2	144	100.0	77.8	77.8	125.0	83.3	97.2	112.5	75.0	100.0
f2	4/4	96	75.0	75.0	75.0	102.1	102.1	102.1	102.1	102.1	102.1
f51m	8/8	1824	82.9	61.8	65.8	84.4	47.6	49.1	88.4	49.9	67.0
misex1	8/7	276	100.0	83.3	91.7	117.4	97.8	117.4	108.7	90.6	108.7
rd53*	5/3	403	41.9	38.7	38.7	26.1	26.1	26.1	26.1	26.1	26.1
rd73*	7/3	2159	29.9	29.1	29.1	20.4	19.5	19.5	20.2	18.5	18.5
rd84*	8/4	5100	21.6	21.2	21.2	14.1	12.7	12.7	20.7	18.5	18.5
sao2	10/4	1392	82.8	60.3	63.8	94.8	60.3	66.8	89.7	61.6	69.1
alu1	12/8	608	—	—	100.0	—	—	118.4	—	—	106.2
alu2	10/8	1904	98.5	60.3	60.3	117.9	57.1	60.7	105.6	55.1	52.0
alu3	10/8	1820	100.0	56.9	56.9	117.7	52.3	54.2	107.1	56.0	47.8
apla	10/12	800	100.0	100.0	100.0	118.8	114.0	118.8	—	—	106.2
clip	9/5	2714	94.9	33.9	36.4	95.1	37.4	47.0	99.5	34.1	44.2
dc1	4/7	135	100.0	88.9	100.0	113.3	88.1	88.1	113.3	88.1	88.1
dc2	8/7	897	100.0	79.5	82.1	117.4	72.2	72.2	108.7	72.5	72.5
dk17	10/11	558	100.0	100.0	100.0	119.4	112.7	119.4	—	—	106.5
dk27	9/9	270	100.0	100.0	100.0	122.2	122.2	122.2	107.4	107.4	107.4
risc	8/31	1316	100.0	96.4	100.0	108.5	104.6	108.5	104.3	100.5	104.3
sqn	7/3	646	94.7	65.8	81.6	100.8	68.3	74.8	100.0	58.8	76.5
wim	4/7	135	100.0	88.9	100.0	100.7	88.1	88.1	100.7	88.1	88.1
xor5*	5/1	176	25.0	25.0	25.0	14.8	14.8	14.8	14.8	14.8	14.8
z4	7/4	1062	81.4	27.1	27.1	91.1	22.8	41.4	82.9	18.8	37.7

Chapter 4

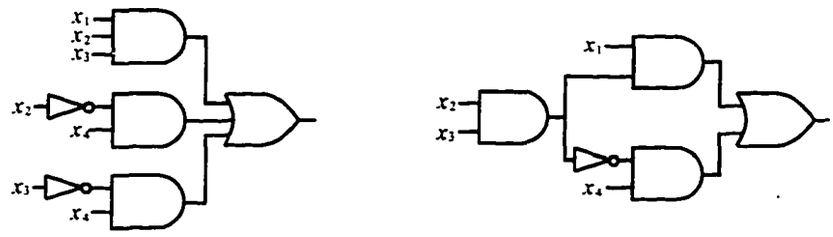
Multi-level Combinational Logic Synthesis

A two-level circuit that is minimal is not necessarily the most efficient in terms of required area or speed. Frequently a multi-level implementation can be found that uses less area and is faster than the two-level realization. Also, there often exists a multi-level circuit with a lower fan-in limit, fewer gates, and fewer interconnections that implements the function.

For example the function $f(x_1, x_2, x_3, x_4) = x_1x_2x_3 + \overline{x}_2x_4 + \overline{x}_3x_4$, which is in its minimal SOP form can be re-expressed as $f(x_1, x_2, x_3, x_4) = x_1(x_2x_3) + x_4(\overline{x_2x_3})$. Figure 4.1 illustrates the circuits corresponding to these two expressions. Note that the multi-level circuit uses one less inverter and two of the gates are smaller than those required in the two-level circuit.

In addition, a system of switching functions requires a multi-level realization if it is to be implemented using gate array or standard cell structures. Indeed, with the vastly expanding availability of Field Programmable Gate Arrays (FPGAs) of various types, there is a corresponding growth in interest in the synthesis of multi-level gate

Figure 4.1: Two and Multi-level Circuit Implementations of $f(x_1, x_2, x_3, x_4)$



level circuits.

Several multi-level circuit synthesis techniques exist and they fall into three basic categories: factoring methods [9, 14, 15], mapping methods [25, 26], and decomposition methods [58, 59, 64, 93].

This chapter provides some background information regarding the various multi-level synthesis procedures and gives some advantages and disadvantages of each. Most attention is paid to decomposition methods since the approach taken in the multi-level synthesis system discussed in the next chapter closely follows that described in [58].

4.1 Factoring Methods

Factoring is an algebraic technique. Factors that are common to several terms of a switching expression are discovered and extracted. The procedure is repeated on the residual expressions until there are no more common factors.

Factoring methods involve the generation of a minimal SOP expression for a function and then factoring the expression to transform it into a multi-level equivalent circuit. For example, the function

$$f(x_1, x_2, x_3, x_4) = x_1x_4 + x_2\overline{x}_3 + x_1x_3 + x_2x_4,$$

can be factored as:

$$f(x_1, x_2, x_3, x_4) = x_1(x_3 + x_4) + x_2(\overline{x}_3 + x_4).$$

SOCRATES [9, 32], MIS [14], ESPRESSO-MLD [8] are some example systems that use factoring to perform multi-level synthesis. These systems have extended the algorithms used in ESPRESSO-II for two-level minimization to the multi-level case.

SOCRATES produces circuits that are targeted to a standard cell implementation and which meet specified timing constraints. MIS produces technology-independent multi-level realizations which are optimized with respect to area and delay constraints. The resulting circuit representations are mapped to a particular technology. ESPRESSO-MLD optimizes multi-level circuits by minimizing an equivalent sequence of 2-level incompletely specified functions.

4.2 Mapping Methods

Mapping methods are synthesis procedures that determines a series of mappings that transforms a function into one that can be implemented using a single logic element. The realization for the circuit consists of a cascade of mapping circuits.

The first step in a mapping procedure is to decompose $f(X)$, $X = \{x_1, x_2, \ldots, x_n\}$ into a mapping circuit M_1 that implements the mapping $X \rightarrow X_1$ and a remainder function $f_1(X_1)$. At each successive stage i, function $f_i(X_i)$ is decomposed into a mapping circuit M_{i+1} implementing $X_i \rightarrow X_{i+1}$ and a remainder function $f_{i+1}(X_{i+1})$. Each mapping stage i introduces some don't cares into the definition of $f_i(X_i)$. The procedure continues until stage m where $f_m(X_m)$ can be implemented trivially by a single logic element.

In most of the developed mapping methods, a limited number of mappings are available depending on the *symmetries* present in the function. The simplest notion

of symmetry means that the function is invariant with respect to the interchange of two variables. A totally-symmetric function is one which is invariant under any permutation of the input variables. A partially symmetric function on the other hand, remains invariant under the permutation of some, but not all, of the inputs. There may be one or more input variable symmetry sets.

The notion of symmetry can be extended, by allowing the interchange of variables or their complements; for example, while $x_1x_2 + x_1x_3 + x_2x_3$ is symmetric in $\{x_1, x_2, x_3\}$, $x_1\overline{x}_2 + x_1x_3 + \overline{x}_2x_3$ is symmetric in $\{x_1, \overline{x}_2, x_3\}$.

Symmetry may be more generally defined to mean that a function is invariant under the interchange of certain assignments to a selection of input variables. This captures the above notions since symmetry in a function with respect to the interchange of x_i and x_j means the function takes the same value for the assignment of $\{0,1\}$ and $\{1,0\}$ to $\{x_i, x_j\}$ for every assignment to the remaining variables. Symmetry with respect to the interchange of x_i and $\overline{x}_j$ is similar, with the assignments being $\{0,0\}$ and $\{1,1\}$. Broader classes of symmetry can thus be defined for variable sets of a given size with respect to a particular input assignment set.

A particular mapping method deals with a certain type, or types, of symmetry. At each stage i, a symmetry is selected to define a mapping circuit and then values are assigned to the don't cares so as to maximize the number of symmetries present in $f_i(X_i)$. This procedure can be quite computationally expensive because of a large number of possibilities in symmetry detection and selection, and in don't care assignment. Additionally, a function may not exhibit a symmetry among any pair of variables, in which case the procedure cannot continue.

In [25, 26], Diaz-Olavarrieta develops a heuristic multi-level synthesis method called the *Goal Oriented Method* which avoids these problems. In this approach the definition of the mapping stages is guided by a simple *goal function*, which is similar to the function to be synthesized and can be implemented using a single logic element.

There is no symmetry detection or don't care assignment performed. The procedure also is not restricted in the selection of mapping functions.

A synthesis technique called the *Direct Implementation Procedure* (DIP) [26], which is developed from the Goal Oriented Method, chooses a goal function, GF, from a set of simple candidate functions. The one selected is the one that differs from f at the fewest *points*. A *point* is one of the 2^n value assignments to the input variables of f. The points at which f and GF differ are called the *essential points* and denoted p_i.

The purpose of the mapping stages is to map any point p_i where $f(p_i) \neq GF(p_i)$ into another point p_j, such that $f(p_i) = GF(p_j)$. The minimum number of essential points mapped by a mapping stage is one. Therefore the maximum number of mapping stages needed is equal to the number of essential points.

The choice of mapping stages and their ordering must be done. Various heuristics are used for this purpose.

4.3 Decomposition Methods

Decomposition is the re-expression of a function as a composition of simpler subfunctions. A multi-level realization is created by determining a sequence of decompositions, each of which expresses a circuit in terms of continually simpler sub-circuits, until the function is entirely expressed in terms of primitive switching elements, usually logic gates.

There are many advantages that decomposition techniques have over other design approaches. First, relatively few temporary results are generated during the computation of the decompositions. This is in contrast to two-level minimization algorithms that require storage of prime implicants for future selection of a minimal cover.

Second, since the possession of a decomposition is a property of the function, it is independent of the set of switching elements that is used to realize the function. Thus decomposition can be adapted quite easily to different switching elements.

Let $f(X)$, $X = \{x_1, x_2, \ldots, x_n\}$ be an n-variable switching function. In general there are two main classes of decomposition: *simple* and *complex* [7, 22]. Simple decompositions [7, 22] are of the form $f(X) = g(h(X_1), X_2)$ where $X_1 \cup X_2 = X$. A *simple disjunctive decomposition* is of the above form but has the condition that $X_1 \cap X_2 = \phi$. A *simple non-disjunctive decomposition* is also of the above form but $X_1 \cap X_2 \neq \phi$.

A complex decomposition [22, 44] is a re-expression of a function as a function of more than two subfunctions. Partition X into subsets $X_1, X_2, \ldots, X_t$, such that

$$X = \bigcup_{i=1}^{t} X_i.$$

A *multiple decomposition* [22] has the general form

$$f(X) = g(h_1(X_1), \ldots, h_t(X_t)).$$

An iterative decomposition has the general form

$$f(X) = g(h_{k-1}(h_{k-2}(\ldots h_2(h_1(X_1), X_2), \ldots), X_{t-1}), X_t).$$

A complex decomposition is a combination of multiple and iterative decompositions. For example, let X be partitioned into subsets X_1, X_2, X_3, and X_4. Then a possible expression for a complex decomposition is

$$f(X) = g(h_2(h_1(X_1), x_2), h_3(X_3), X_4).$$

If $X_i \cap X_j = \phi$ for all $i \neq j$ then the decomposition is a *complex disjunctive decomposition.* If some $X_i \cap X_j \neq \phi$ for some $i \neq j$, then the decomposition is a *complex non-disjunctive decomposition.* g is called the *image function* of the decomposition.

There are three steps involved in the synthesis of circuits by decomposition: identification of possible decompositions, selection of the decompositions, and construction of a sequence of decompositions representing the desired function.

Ashenhurst [7] presented the first major study of decomposition by considering the problem of finding the simple disjunctive decompositions of completely specified functions. Ashenhurst's approach requires $f(X)$ to be represented by a partition matrix with the variables in X_1 labeling the columns and those in X_2 labeling the rows.

The *column multiplicity* of a partition matrix, denoted by μ, is the number of distinct columns in the partition matrix. Ashenhurst shows in [7] that $f(X)$ has a simple disjunctive decomposition if and only if the corresponding partition matrix has $\mu \leq 2$. There are $2^n - n - 2$ possible partition matrices to consider for a function of n variables.

Since g and h cannot have any common arguments, the simple disjunctive decomposition is rather restrictive. A simple non-disjunctive decomposition allows g and h to have common arguments. Curtis [22] was the first to present a method for determining the simple non-disjunctive decompositions in a completely specified switching function.

Complex disjunctive decompositions are determined by applying several theoretical results to the set of simple decompositions of a function. Curtis [22] and Karp [44] present the main results for completely specified switching functions.

Hight [35] extends Ashenhurst's technique to incompletely specified functions.

Definition 4.1 *Given $f(X)$, two assignments (a_i, a_j) and (b_i, b_j) to the pair of variables x_i and x_j, $1 \leq i, j \leq n$, are* **compatible** *with respect to $f(X)$ if*

$$f_a(x_1, \ldots, a_i, \ldots, a_j, \ldots, x_n) = f_b(x_1, \ldots, b_i, \ldots, b_j, \ldots, x_n),$$

for all assignments to the variables in $X - \{x_i, x_j\}$ *for which both* f_a *and* f_b *are defined. Compatibility is denoted as* $a_i a_j \sim b_i b_j$.

For example, $01 \sim 11$ for variables x_2 and x_1 in the function

		x_2x_1			
		00	01	10	11
x_3	0	0	–	1	1
	1	0	1	0	1

Two columns of a partition matrix are compatible if the two corresponding assignments to the variables labeling the columns are compatible. Hight [35] shows that an incompletely specified function has a simple disjunctive decomposition if, and only if, the columns of the corresponding partition matrix can be divided into two sets of mutually compatible columns. A set of columns C is mutually compatible, if, for every $c_i, c_j \in C, c_i \sim c_j, i \neq j$. Hight's result is a special case of a result due to Roth and Karp [77].

In [58] Miller extends Roth and Karp's algorithm [77] and an implementation due to Karp et al. [45] to the multiple output case and develops a new approach to combinational switching circuit synthesis. This approach is applied in [59] in the area of synthesizing more easily testable circuits.

More recently, decomposition techniques have been applied to PLA decomposition [24, 83]. Also various spectral techniques have been applied in detecting disjoint decompositions in completely specified functions [75, 92, 93]

This dissertation deals mainly with the Roth-Karp [77] approach and methods discussed in [58].

4.4 Summary

This chapter provides a brief summary of the three main approaches to the synthesis of multi-level switching circuits: factoring, mapping, and decomposition. Function decomposition, is the subject of the following chapters.

Chapter 5 provides a summary of an algorithm developed by Miller [58] and describes enhancements to it that enable it to produce results for a broader range of functions.

Chapter 6 describes the use of autocorrelation techniques to extend the capabilities of the two-place decomposition to functions that do not exhibit symmetries.

Chapter 7 presents a new technique for multi-level synthesis that is targeted to two-place decomposition. This approach can be used to handle non-two-place decomposable functions and to synthesize large functions using the two-place decomposition method.

Chapter 5

Two-Place Decomposition

The most general type of decomposition is the complex decomposition as described in Section 4.3. Implementing an algorithm to produce decompositions in their full generality is a complex problem. Difficulties arise in the detection of decompositions. selection of a *best* decomposition and in assignment of functions to the h_i subfunctions. Also decomposition of incompletely specified functions is difficult.

Therefore, to make decomposition steps more manageable and to develop usable algorithms for the synthesis of multi-level circuits, the scope of the decomposition problem is here restricted to the class of *two-place decompositions* where $1 \leq t \leq 2$ and $|X_1| = 2$.

This reduced decomposition problem is addressed in a two-place decomposition algorithm developed by Miller [58, 59, 64]. It produces a multi-level circuit composed of one and two-input logic gates. The procedure also deals quite well with incompletely specified functions.

This chapter summarizes the algorithm and describes enhancements that enable the procedure to determine a decomposition for some functions that the original algorithm is unable to handle.

Section 5.1 outlines the two-place decomposition procedure of [58]. In Section 5.2 the decomposition of an example function is detailed. Sections 5.3 to 5.3.2 discuss various aspects of the procedure and provide some enhancements to it. Enhancements are made in the areas of decomposition selection and mapping function selection. The improved decomposition algorithm, called **Decomp** is compared to the original to gauge the effect of the enhancements that are introduced. The remaining sections compare results produced by the decomposition procedure with results published in the literature. Further enhancements employing autocorrelation techniques are presented in Chapter 6.

5.1 Two-Place Decomposition

Given an n-place function $f(X)$, $X = \{x_1, x_2, \ldots, x_n\}$, a two-place decomposition is an expression of the form

$$f(X) = g(h_1(x_i, x_j), h_2(x_i, x_j), X - \{x_i, x_j\})$$

where $1 \leq i, j \leq n$ and h_1 and h_2 are functions of at most two variables and g is the *image function* of the decompositions. Each iteration in a decomposition procedure operates upon the image function produced by the previous iteration.

Two-place decompositions fall into three classes: *simple disjunctive*, *simple non-disjunctive*, and *complex disjunctive*.

A simple disjunctive (SD) decomposition is of the form

$$f(X) = g(h(x_i, x_j), X - \{x_i, x_j\})$$

where g is a function of $n-1$ variables. This decomposition is termed simple because only one h is used and disjunctive since h and g have no common variables.

A simple non-disjunctive (SND) decomposition is of the form

$$f(X) = g(h(x_i, x_j), x_k, X - \{x_i, x_j\})$$

where x_k is either x_i or x_j. It is non-disjunctive since h and g have a variable in common.

A complex disjunctive (CD) decomposition is of the form

$$f(X) = g(h_1(x_i, x_j), h_2(x_i, x_j), X - \{x_i, x_j\})$$

There are no common variables between g and the h functions.

A decomposition is *non-trivial* if g is defined for fewer assignments to its input variables than is $f(X)$. This can arise when g has fewer input variables than $f(X)$, or the same number of input variables but more assignments for which the output value is a don't care. Only non-trivial decompositions are of use since they reduce the complexity of the image function.

A *trivial* decomposition simply relabels input assignments. For example, consider a simple disjunctive decomposition of f in variables x_i and x_j. If $h_1 = x_i \ominus x_j$ and $h_2 = x_i$, then the result is the trivial relabeling

x_i	x_j	h_1	h_2
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Thus a simple non-disjunctive decomposition using XOR is not considered.

However, this relabeling corresponds to a Type 4 linear translation, which is used in the linearization process described in Chapter 2. Chapter 6 provides a detailed discussion of linearization and its use in enhancements to the two-place decomposition procedure. As seen in Chapter 6, two-place decomposition can benefit from linearization.

The steps performed by the two-place decomposition procedure are: identification of possible two-place decompositions, selection of appropriate two-place decompositions. and application of the selected decompositions to produce an image function. These steps are recursively repeated on the newly produced image function until an image function is produced that is composed of a single switching element.

5.1.1 Identification of Two-Place Decompositions

The two-place decompositions that are present in a switching function are identified by certain two-place symmetries that are exhibited in pairs of variables. A two-place symmetry exists in variables x_i and x_j, if f is invariant under the interchange of two assignments to x_i and x_j. There are four possible assignments to the pair x_i and x_j. Thus for each pair of variables there are six possible two-place symmetries. These symmetries are detailed in [29, 64]

Visually one may use a *decomposition chart* [22] to detect if a function exhibits any compatible assignments to a pair of variables x_i and x_j. With respect to two-place decomposition. a decomposition chart is a partition matrix with four columns and 2^{n-2} rows. The columns are identified by the four possible assignments to x_i and x_j. The rows are identified by all the assignments to the remaining variables. Figure 5.1b provides the decomposition charts for the function given in Figure 5.1a. By inspection of these charts one can see the following compatible assignments to pairs of variables:

x_4x_3: $00 \sim 10$
x_4x_2: $00 \sim 10$
x_3x_2: $00 \sim 01$, $01 \sim 10$, $10 \sim 00$

Use of decomposition charts is impractical for functions with large numbers of inputs since there are $\binom{n}{2} = \frac{n(n-1)}{2}$ decomposition charts to examine and an exhaustive

comparison must be done between the appropriate columns of each chart. Also the full truth table for the function is required.

A much more efficient method for identifying the symmetries present is the one described in [58, 64]. This method accepts the function in conventional cube notation (as used by programs like ESPRESSO) and determines the symmetry information for all pairs of variables in one processing of the cubes. It is expedient to make the cube list as short as possible, *i.e.* minimize the list, as each pair of cubes must be compared to determine the symmetries.

The two-place symmetries are identified by comparing pairs of on and off cubes: don't cares are omitted from the cube list. Only pairs of cubes that are Hamming distance 1 or 2 apart are considered. The method compares all on and off cube pairs and uses cube-based operations to form tables that encode the pair's symmetry information. Refer to [58, 59] for the details of this process.

The symmetry information for a pair of variables x_i and x_j, is analyzed to identify which decompositions are possible. For any pair x_i and x_j, there are 64 possible compatibility relationships among the assignments to x_i and x_j. However, the majority of these relationships identify trivial decompositions, or they indicate that the function is independent of one or both of x_i and x_j, or they represent multiple decompositions. Thus the number of compatibility relationships that identify decompositions is relatively small.

Table 5.1 indicates which decompositions exist in a function when these compatibility relationships are exhibited in x_i and x_j. The table also provides the mapping functions that can be used for h_1 and h_2 [58, 59]. The complement of h_1 or h_2 may also be used.

The decomposition identification process must consider the fact that compatibility is not transitive. For example, the decomposition chart

Table 5.1: Two-Place Symmetries and Possible Decompositions in x_i and x_j
SD - Simple Disjunctive
SND - Simple Non-Disjunctive
CD - Complex Disjunctive

Two-Place Symmetries	Decomposition Exhibited	h_1 and h_2
$00 \sim 01,\ 00 \sim 10,\ 01 \sim 10$	SD	$h_1 = x_i x_j$
$00 \sim 01,\ 00 \sim 11,\ 01 \sim 11$	SD	$h_1 = x_i \overline{x}_j$
$00 \sim 10,\ 00 \sim 11,\ 10 \sim 11$	SD	$h_1 = \overline{x}_i x_j$
$01 \sim 10,\ 01 \sim 11,\ 10 \sim 11$	SD	$h_1 = \overline{x}_i \overline{x}_j$
$00 \sim 11,\ 01 \sim 10$	SD (XOR)	$h_1 = x_i \oplus x_j$
$00 \sim 01$	SND	$h_1 = x_i x_j$ or $h_1 = x_i \overline{x}_j$, $h_2 = x_i$
$00 \sim 10$	SND	$h_1 = x_i x_j$ or $h_1 = \overline{x}_i x_j$, $h_2 = x_j$
$01 \sim 11$	SND	$h_1 = x_i \overline{x}_j$ or $h_1 = \overline{x}_i \overline{x}_j$, $h_2 = x_j$
$10 \sim 11$	SND	$h_1 = \overline{x}_i x_j$ or $h_1 = \overline{x}_i \overline{x}_j$, $h_2 = x_i$
$00 \sim 11$	CD	$h_1 = \overline{x}_i x_j$, $h_2 = x_i \overline{x}_j$, or, $h_1 = x_i \oplus x_j$, $h_2 = \overline{x}_i x_j$ or $h_2 = x_i \overline{x}_j$
$01 \sim 10$	CD	$h_1 = x_i x_j$, $h_2 = x_i + x_j$, or, $h_1 = x_i \oplus x_j$, $h_2 = x_i x_j$ or $h_2 = \overline{x}_i \overline{x}_j$

		x_2x_1 = 00	01	10	11
x_3	0	0	–	1	0
	1	1	1	1	0

indicates that for variables x_2 and x_1, $00 \sim 01$ and $01 \sim 10$. Therefore a simple non-disjunctive decomposition, and a complex disjunctive decomposition exists in x_1 and x_2. However, since $00 \not\sim 10$, a simple-disjunctive decomposition does not exist.

For a system of switching functions, the potential two-place decompositions for each output function are identified in turn. The decompositions from the individual output functions are merged to allow for sharing of gates in the final realization.

In the algorithm of [58] decompositions are combined in a fixed lexicographic order and mergers are accepted as they arise. The algorithm attempts to identify those mergers involving as many output functions as possible *i.e.* to share as many gates as possible between circuit outputs.

It should be noted that the merged decompositions do not have to be of the same type, but must of course involve the same variables. For example, a complex decomposition employing $h_1 = x_i x_j$ and $h_2 = x_1 + x_j$ can be combined with simple disjunctive decompositions using either $h_1 = x_i x_j$ or $h_1 = x_i + x_j$. In this case the cost of the merged decomposition is the same as the cost of the complex decomposition alone.

5.1.2 Selection of Two-Place Decompositions

Once any possible mergers are made, decompositions are chosen from those available using a simple set of heuristics. Each chosen decomposition is one of lowest cost. determined by the simplicity of the decomposition and from the number of gating levels in the circuit. The set of heuristics is as follows [58]:

1. choose the decomposition which is applicable to the greatest number of output functions;
2. in the case of more than one such decomposition, choose a simple disjunctive decomposition in preference to any other type, and choose a simple non-disjunctive decomposition in preference to a complex disjunctive one;
3. if there is more than one, choose the one where the number of gating levels between the primary circuit inputs and the gates created by the decomposition is as low as possible;
4. if there is still more than one decomposition to choose from, the choice is made arbitrarily (in the current implementation it is one whose primary input numbering is lowest or involving the earlier created gate outputs).

Each of the available decompositions is applicable to a subset of the m outputs of $f(X)$. After the first decomposition is selected, the procedure attempts to select

another decomposition using the above selection heuristics with the added restriction that no newly chosen decomposition affects any function output already affected by a chosen decomposition. This restriction avoids possible inconsistencies in the image function caused by the application of two decompositions to the same function.

The selection process continues until no more outputs can be affected or until no more decompositions exist.

5.1.3 Application of Two-Place Decompositions

The functions that are assigned to each of h_1 and h_2 are fixed (up to complementation) for each type of decomposition. For each decomposition chosen by the above selection process, the corresponding functions are assigned to h_1 and h_2, and the image function of the decomposition is found by a straightforward substitution process.

After decompositions have been identified and possibly merged, selected, and finally applied to form the image function, the whole process is then repeated on that image function. The entire procedure iterates until a final image function corresponding to a single logic gate has been identified for every output function. Note that for the case of a system of m functions, the decompositions chosen create an image function for certain of the output functions while the others are not affected. These latter functions pass to the next decomposition step unaltered.

5.2 Decomposition Procedure Example

This section provides details of the decomposition of an example function. Decomposition charts are used here to detect function symmetries because the charts are easily understood and one can quickly recognize a symmetry simply by visual inspection.

Figure 5.1: Function for Example Decomposition

x_4	x_3	x_2	x_1	f
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0

x_4	x_3	x_2	x_1	f
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

(a)

x_2x_1 \ x_4x_3	00	01	10	11
00	0	0	0	0
01	1	1	1	1
10	0	0	0	1
11	1	0	1	1

x_3x_1 \ x_4x_2	00	01	10	11
00	0	0	0	0
01	1	1	1	1
10	0	0	0	1
11	1	0	1	1

x_4x_1 \ x_3x_2	00	01	10	11
00	0	0	0	0
01	1	1	1	0
10	0	0	0	1
11	1	1	1	1

(b)

In the actual implementation of the algorithm, symmetry detection is performed as summarized in Section 5.1.1 and detailed in [58, 64].

Consider the function $f(x_1, x_2, x_3, x_4)$ defined by the truth table of Figure 5.1(a). Upon examination of the decomposition charts for f, which are provided in Figure 5.1(b), the following symmetries and decompositions are identified.

x_4x_3:	$00 \sim 10$	Simple non-disjunctive
x_4x_2:	$00 \sim 10$	Simple non-disjunctive
x_3x_2:	$00 \sim 01$, $01 \sim 10$, $10 \sim 00$	Simple Disjunctive

Choose the simple disjunctive decomposition involving variables x_3 and x_2 since the decomposition selection heuristics state that a simpler decomposition should be chosen before a more complex one. The choice of this decomposition results in the replacement of the two variables x_3 and x_2 by one new variable. Thus g depends on one variable less than does f and is therefore much simpler.

Figure 5.2: Image Function g_1

h_1	x_4	x_1	g_1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(a)

h_1 \ x_4x_1	00	01	10	11
0	0	1	0	1
1	0	0	1	1

x_1 \ h_1x_4	00	01	10	11
0	0	0	0	1
1	1	1	0	1

x_4 \ h_1x_1	00	01	10	11
0	0	1	0	0
1	0	1	1	1

(b)

Define $h_1 = x_3x_2$. Thus $f = g_1(h_1(x_3x_2), x_4, x_1)$, where image function g_1 is defined in Figure 5.2(a). Inspection of g_1's decomposition charts in Figure 5.2(b) identifies the following symmetries and decompositions that are exhibited.

h_1x_4: $00 \sim 01$ Simple non-disjunctive
h_1x_1: $10 \sim 11$ Simple non-disjunctive

Since the decompositions that exist for g_1 are of the same complexity, and both come through the same number of gating levels, choose one decomposition arbitrarily. So choose the decomposition involving variables h_1 and x_4 and define $h_2 = h_1x_4$. This produces $g_1 = g_2(h_2(h_1, x_4), h_1, x_1)$. The truth table and decomposition charts for g_2 are provided in Figure 5.3. Notice that no variables are removed from the problem by applying a simple non-disjunctive decomposition. However don't-cares are introduced, resulting in the simpler image function g_2.

The decompositions present in g_2 are:

h_1x_1: $00 \sim 10$, $10 \sim 11$, $11 \sim 00$ Simple disjunctive
h_2x_1: $10 \sim 11$ Simple non-disjunctive

Note that the assignment 10 to variables h_2 and h_1 is compatible with all other assignments to that pair. Any decomposition involving this pair of variables would

Figure 5.3: Image Function g_2

h_2	h_1	x_1	g_2
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	–
1	0	1	–
1	1	0	1
1	1	1	1

(a)

h_2 \ h_1x_1	00	01	10	11
0	0	1	0	0
1	–	–	1	1

h_1 \ h_2x_1	00	01	10	11
0	0	1	–	–
1	0	0	1	1

x_1 \ h_2h_1	00	01	10	11
0	0	0	–	1
1	1	0	–	1

(b)

be a trivial one serving only to relabel their input assignments and would not decrease the complexity of the image function.

Choose the simple-disjunctive decomposition in h_1 and x_1 and define $h_3 = \bar{h}_1 x_1$. This again removes a variable from the problem yielding $g_2 = g_3(h_3(h_1, x_1), h_2)$. Figure 5.4 depicts image function g_3, which is a two variable function and thus can be implemented as $g_3 = h_3 + h_2$ or as $g_3 = h_3 \oplus h_2$. The output of this last stage is the value of f.

The resulting circuit that implements f is given in Figure 5.5.

Figure 5.4: Image Function g_3

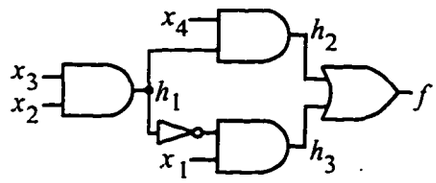
h_3	h_2	g_3
0	0	0
0	1	1
1	0	1
1	1	–

(a)

h_3 \ h_2	0	1
0	0	1
1	1	–

(b)

Figure 5.5: Circuit Implementing $f(x_1, x_2, x_3, x_4)$



5.3 Effects of Decomposition Choices

The existence of decompositions in an image function is affected by which decomposition is selected out of those that are available and by which mapping functions for h_1 and h_2 are utilized for the chosen decomposition. *Better* choices of decompositions and mappings are ones that create or maintain simpler decompositions in the image function.

In [58, 59] a set of simple heuristics is used to guide the choice of decomposition to apply. These heuristics do not consider how the choice impacts on the image function with respect to the number and types of decompositions that exist. So it is possible to choose a decomposition that yields an image function that has no decompositions or one that has no simple ones.

5.3.1 Decomposition Selection

For a system of m switching functions there may be several decompositions from which to choose at each step of the decomposition procedure. To aid in making a good choice of decomposition, a measure of merit is computed using the selection heuristics described in Section 5.1.2. These heuristics favour decompositions that affect the most function outputs. The merging process attempts to maximize this number for a decomposition.

In the algorithm of [58] the method of merging decompositions is a greedy one *i.e.*

decompositions are combined in a fixed lexicographic order and mergers are accepted as they arise. Because of this there may be some decompositions that are not merged or are merged in a non-optimal way.

5.3.2 Mapping Selection

The choice of mapping also has an impact on the image function. For each decomposition type, Miller [58, 59] identifies several different mapping functions that can be used for h_1 and h_2. These are provided in Table 5.1. There are:

- one possible assignment to h_1 for each of the five simple disjunctive decompositions;
- two possible assignments to h_1 and one possible assignment to h_2, for each of the four simple non-disjunctive decompositions;
- one assignment to each of h_1 and h_2 for the complex disjunctive decompositions. In addition if the alternate mapping functions using the XOR function are considered, there are an additional two possibilities for a total of three for each of the two complex disjunctive decompositions.

Additionally the complement of either h_1 or h_2 may be used. This greatly increases the number of possible mapping function assignments from which to choose. Fortunately, complementation does not affect the existence of decompositions in the image function, so the complements need not be considered.

As an example of how the choice of mapping can affect the existence of future decompositions, consider the function f and the list of its symmetries provided in Figure 5.6. From these symmetries, f has only simple non-disjunctive decompositions in the corresponding variable pairs.

Figure 5.6: Original f and Existing Symmetries

x_4x_3 \ x_2x_1	00	01	10	11
00	1	1	1	1
01	0	0	1	0
10	1	1	0	0
11	0	0	0	0

symmetries
x_4x_2: 00 ~ 10
x_3x_1: 00 ~ 01
x_4x_1: 10 ~ 11
x_2x_1: 00 ~ 01

Assume that the simple non-disjunctive decomposition in variables x_2 and x_1 is chosen. Figure 5.7 provides the two image functions that are obtained from the two possible mapping assignments to h_1 and h_2. Image function g_1 uses mapping assignments $h_1 = x_2x_1$ and $h_2 = x_2$. The mapping $h_1 = \overline{x}_2 + x_1$ and $h_2 = x_2$ is used for image function g_2. Note that g_1 exhibits no simple disjunctive decomposition while g_2 exhibits two simple disjunctive decompositions: in variables x_4 and x_2, and in variables x_3 and h_1

Good choices of mapping can have a significant effect not only on the decomposability of a function but also on the size of the resulting circuit. Table 5.2 exemplifies

Figure 5.7: Image Functions of f Corresponding to Choice of h_1

g_1

simple non-disjunctive $h_1 = x_2x_1$

x_4x_3 \ x_2h_1	00	01	10	11
00	1	–	1	1
01	0	–	1	0
10	1	–	0	0
11	0	–	0	0

symmetries
x_4x_2: 00 ~ 10
x_3h_1: 00 ~ 01
10 ~ 01
x_4h_1: 10 ~ 11

g_2

simple non-disjunctive $h_1 = \overline{x}_2 + x_1$

x_4x_3 \ x_2h_1	00	01	10	11
00	–	1	1	1
01	–	0	1	0
10	–	1	0	0
11	–	0	0	0

symmetries
x_3x_2: 10 ~ 11
x_4x_2: 00 ~ 01
00 ~ 10
01 ~ 10
x_3h_1: 00 ~ 01
00 ~ 10
01 ~ 10
x_4h_1: 10 ~ 11

Table 5.2: CDNES Decomposition Mapping Function Comparison

	$h_1 = x_i x_j$ $h_2 = x_i + x_j$				$h_1 = x_i x_j$ $h_2 = x_i \oplus x_j$				$h_1 = x_i + x_j$ $h_2 = x_i \oplus x_j$			
func	gt	lit	trn	lvl	gt	lit	trn	lvl	gt	lit	trn	lvl
9sym	70	129	352	18	30	90	180	8	—	—	—	—
add6	32	86	192	11	27	76	162	11	37	86	182	21
rd53	20	47	116	7	16	44	96	6	15	40	82	7
rd73	43	97	246	11	22	62	132	6	—	—	—	—
rd84	62	131	344	14	28	80	168	6	—	—	—	—
z4ml	18	48	108	7	15	42	90	7	21	48	102	12

this for the non-equivalence-symmetry complex-disjunctive (CDNES) decomposition mappings. For the CDNES decomposition there are three possible mappings, two of which involve the XOR function. These mappings are the AND/OR gate combination. the AND/XOR combination and the OR/XOR combination.

For each function in Table 5.2 the decomposition procedure is performed three times, each using a different mapping for a CDNES decomposition. The table provides the number of gates, literals, CMOS transistors, and gating levels for the circuit generated using each mapping. As can be seen the mapping used can have a substantial effect on the size of the resulting circuit implementing a function.

Function *9sym* is an excellent example of how the mapping function choice affects the result. With the AND/OR mapping the realization has 70 gates and 18 gating levels but with AND/XOR the realization has 30 gates and 8 gating levels. Use of the OR/XOR mapping, leads to an image function that has no decompositions.

The goal is always to make the image function simpler. Since only non-trivial decompositions are selected, applying a selected decomposition always reaches this goal in that an input variable is removed or don't cares are introduced. However, part of the goal should be to produce or maintain simpler decompositions in the image function.

Algorithm 5.1: Enhanced Decomposition Merge Algorithm

```
Repeat until no decompositions remain
    {
    compute the values of merit for all possible mergers
    select the merger of highest merit
    perform the actual merge operation
    remove the merged decompositions from the decomposition list
    }
```

5.4 Enhancement Descriptions

Enhancements are made to the original two-place decomposition procedure developed by Miller [58] in the area of decomposition merging and selection, and in mapping selection for CDNES decompositions. Also a post-processing circuit reduction step is added to combine gates into larger CMOS-type circuit elements.

In the original procedure, decompositions are combined in a "greedy" fashion in that detected decompositions are merged as they arise with the first possible decomposition found on a list of decompositions. Because of this, decompositions may not be merged in an optimal way.

Therefore the merging procedure is enhanced to merge the decompositions only after all existing decompositions have been detected for a pair of variables. Algorithm 5.1 provides the algorithm for the new merge procedure. The merging process depends on the *values of merit* computed for the various possible mergers. This is a measure of "goodness" of the merger and is the sum of the individual decomposition values of merit. The value of merit for a decomposition is computed from the simplicity of the decomposition and from the number of output functions to which it is applicable.

Each time a merge is performed the values of merit are recomputed for the remain-

ing decompositions. This is necessary because the merits no longer reflect reality when the next iteration occurs and the relationships among the decompositions change with the removal of some of them.

For the CDNES decomposition mapping selection, the user either chooses which mapping to use or lets Decomp make a choice based on the values of merit that are computed in the merger process. If Decomp is given the responsibility of choosing, it bases its choice on the value of merit that is computed for each of the possible mappings. There is no change made to Algorithm 5.1 to perform this operation. The only thing done is to set the CDNES mapping functions to a "dummy" function that merges with any decomposition appropriate for the CDNES decomposition.

Algorithm 5.1 allows this type of choice without requiring any changes to the algorithm itself. The original merge algorithm could not handle this type of flexibility.

5.4.1 Circuit Reduction Post-processing

In the switching circuit that Decomp generates, all of the mapping stages are implemented with two-input gates and inverters. After the circuit is generated one can do some physical reduction of that circuit and much of it can be done by a simple inspection process. The "-r" command option to Decomp (see Appendix A) is used to enable this feature. The rules that are followed by Decomp are based on inspection and the Boolean identities and are targeted to CMOS circuits. These rules are:

- If both the uncomplemented and complemented outputs of a gate are required, then generate the two signals using a NAND/NOR gate, for the complemented output, followed by an inverter, for the uncomplemented output. This requires two fewer transistors than if an AND/OR and inverter combination are used.
- Any of the inputs of an XOR gate can be freely complemented as well as the polarity changed to XNOR without changing the circuit cost because both XOR

and XNOR use the same number of transistors.

- If the only gate to which a positive gate output goes is an inverter, then the inverter is moved up and the positive gate is complemented to yield the same function with two fewer transistors and one fewer level. If the inverter is on the longest path of the circuit then the delay of the circuit is made shorter.
- Fanout free portions in the circuit can be combined into larger circuit elements such as multi-input AND/NAND or OR/NOR gates, or into more complex structures such as AOIs and OAIs that are commonly implemented in CMOS. This process can significantly reduce the number of transistors and the number of gating levels to compute the same subfunction.

All of the simplifications are percolated from outputs to inputs. Significant improvements can be made by performing this procedure on several benchmark functions as shown in the tables in the next section. The reader may refer to Appendix E for some example circuits produced by the post-processing operation.

This post-processing step could also be extended to generalized technology mapping onto standard cells, FPGAs, or other sea-of-gates structures with a correspondingly more complex procedure.

5.4.2 Example Circuits

This section provides some circuits that are generated by the decomposition procedure for a few simple problems.

Figure 5.8 provides the circuit generated by the decomposition procedure for the "two out of five" code validity checker. Its output is one when exactly two of its inputs are one.

Figure 5.8: Circuit for the 2-out-of-5 Checker

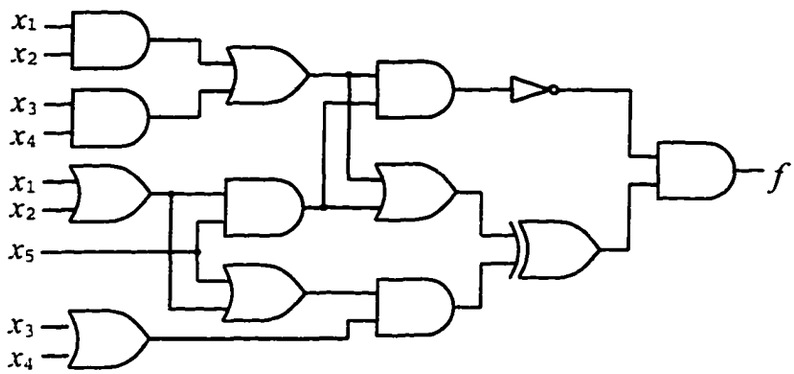


Figure 5.9 shows the circuit generated for the two-bit multiplier. The circuit has four inputs and four outputs and produces the four-bit product of its two two-bit input values a and b.

Figure 5.10 depicts the best circuit generated by the decomposition procedure for a 6-bit adder, which has twelve inputs and seven outputs. It produces the seven-bit sum s, of two six-bit values a and b.

This circuit is generated using the AND/XOR mapping for a CDNES decomposition. It has the standard ripple carry adder structure in which each stage is a full adder. The circuit for the full adder is shown in Figure 5.11. A total of 27 gates

Figure 5.9: Circuit for the Two-bit Multiplier

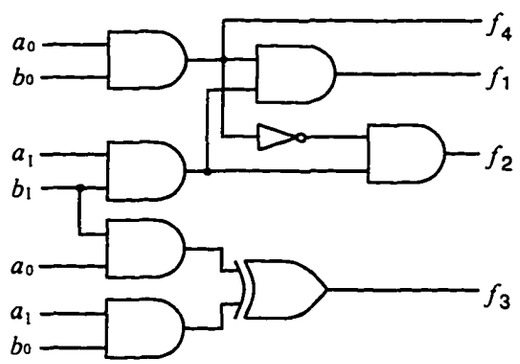


Figure 5.10: Circuit for the Six-bit Adder

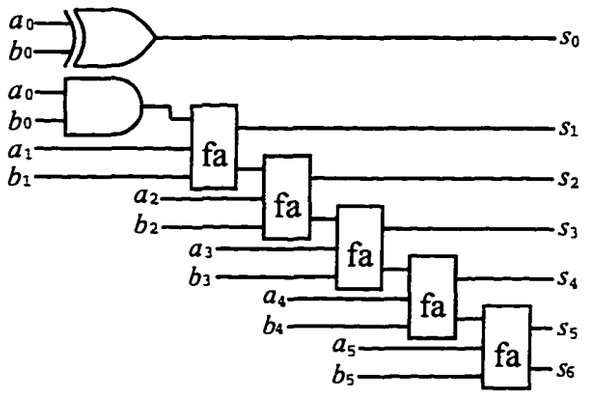
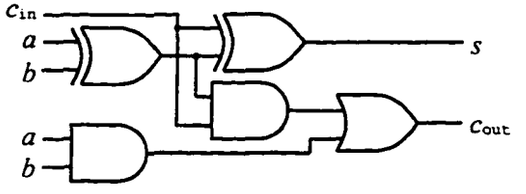


Figure 5.11: Circuit for the Full Adder



and 11 gating levels are used in the adder circuit. If the AND/OR mapping is used, the resulting circuit has 32 gates and 11 gating levels, and if the OR/XOR mapping is used, it has 37 gates and 21 gating levels. These results again show the effect of mapping choice on the size of a realization.

5.5 Multi-level Synthesis System Comparisons

In this section the two-place decomposition procedure (Decomp) is compared with two other multi-level synthesis systems: Oasis [15] and MIS [14]. Factoring is the basis of the multi-level synthesis procedures used in these systems. Realization size is the primary measure for comparison. The execution time required by each system to synthesize a realization is also considered.

For comparing Decomp with MIS, MIS is used to perform technology mapping operations. For comparisons with Oasis five multi-level realizations are generated for certain benchmark functions. The results provided in the tables for each realization are divided into columns as follows:

gt number of gates required

lit number of literals

trn number of CMOS transistors required

lvl number of gating levels

sec the time in seconds on a Sun SPARCstation 2 required to obtain the realization (as reported by system timing facilities).

Results are provided for the realizations generated by:

1. *Decaf*, Oasis' decomposition and factoring utility. Decaf factors and decomposes the input function into a set of subfunctions which are mapped by Oasis' technology mapper Mcmap [56] to the Oasis technology library to produce the final multi-level circuit.

2. *Decomp*, the enhanced two-place decomposition procedure. Here, the mapping functions used for CDNES decompositions are not fixed. Rather, Decomp selects the appropriate mappings for CDNES decompositions based on other decompositions that exist.

3. *Decomp-r*, Decomp with circuit reduction post-processing.

4. *Recomp*, a recomposition procedure developed by Walsh [97]. Recomp analyzes the circuit produced by Decomp to combine portions into larger circuit elements.

5. *Mcmap* to map the net list produced by Decomp onto the Oasis technology library.

The following sections present the comparison results using various functions, which have been divided into three categories:

1. functions that are totally symmetric;
2. functions that are not totally symmetric but tend to exhibit substantial symmetry, such as arithmetic functions;
3. functions that exhibit little symmetry, such as control functions.

5.5.1 Symmetric Functions

Comparison with Oasis

Tables 5.3(a) and 5.3(b) compare Decomp's results with those of Oasis [15, 56] for a set of symmetric functions.

As seen in Table 5.3(a), for these functions, Decomp produces realizations that are typically less than half the size of those produced by Decaf. However when the Decomp generated realizations are mapped onto the Oasis technology library using Mcmap, the resulting realizations are larger in terms of gates, literals and levels, but are slightly smaller with respect to transistor count.

A similar trend is seen for Recomp in Table 5.3(b). However Decomp-r generally reduces the sizes of the circuits in all respects except in terms of levels. But, it doesn't increase the number of levels.

Another observation is that Decomp generates its realizations anywhere from three to 42 times faster than Oasis does. Also if Decomp performs its simple post-processing technology step, little extra time is required to reduce the size of the realizations whereas Mcmap and Recomp require significantly more time.

Table 5.3: Comparison of Symmetric Functions with Oasis
(a) Oasis, Decomp, Mcmap
(b) Decomp-r and Recomp technology mapping

	Decaf					Decomp					Mcmap				
func	gt	lit	trn	lvl	sec	gt	lit	trn	lvl	sec	gt	lit	trn	lvl	sec
2of5	15	33	66	6	4.2	13	27	74	5	0.1	18	48	82	7	5.0
9sym	179	385	770	16	8.6	30	90	180	8	2.7	42	102	178	9	5.6
rd53	34	75	146	7	4.6	16	44	96	6	0.2	23	51	90	6	5.1
rd73	101	225	446	12	8.0	22	62	132	6	1.5	23	63	108	6	5.4
rd84	180	387	766	13	16.8	28	80	168	6	5.3	40	92	164	8	5.5
xor5	4	16	24	3	4.1	4	16	24	3	0.1	4	16	24	3	4.8
z4ml	25	55	108	5	4.7	15	42	90	7	0.6	16	43	74	5	5.2

(a)

	Decomp-r					Recomp				
func	gt	lit	trn	lvl	sec	gt	lit	trn	lvl	sec
2of5	9	21	45	5	0.1	18	48	80	8	3.6
9sym	30	90	156	8	2.6	32	92	156	8	3.4
rd53	14	42	80	5	0.2	15	43	74	6	3.3
rd73	20	60	106	6	1.6	18	58	98	6	3.3
rd84	27	79	150	6	5.3	32	80	138	5	3.7
xor5	4	16	24	3	0.1	6	18	28	3	2.9
z4ml	15	42	72	7	0.6	15	42	72	6	3.0

(b)

Comparison with MIS

Table 5.4 provides sizes for multi-level realizations generated by MIS [14] and by Kim and Dietmeyer [51] for a set of symmetric functions. The numbers that are listed give the counts of the literals in the multilevel realization of the functions.

The first three columns provide different researchers' reported literal counts for realizations generated by MIS. These literals are counted before a mapping to a technology is made. The first column lists the values reported in [51]. Results in the column labeled "Tai" are reported in [87]. The third column lists the best results obtained by the author.

There is a significant difference among the various reported results, the most

Table 5.4: Comparison of Symmetric Functions with MIS and Dietmeyer's Recode
— - result not provided

	MIS Before Mapping			Diet's	After Mapping			
func	Diet	Tai	RT	Recode	MIS 2	decomp	MIS n	decomp n
2of5	—	—	24	—	44	27	36	24
9sym	260	—	73	58	128	90	105	102
rd53	40	52	38	29	57	44	54	43
rd73	143	118	76	48	135	62	105	63
rd84	—	—	154	—	270	84	209	91
z4ml	38	—	37	36	67	42	49	43
xor5	—	—	16	—	19	16	16	16

marked being for the functions *9sym* and *rd73*. The best results are obtained by attempting various MIS operations to obtain lower literal counts and iterating until no decrease in the counts occurs. This is an entirely manual procedure and is fairly time consuming but better results than those reported are achieved.

The fourth column gives the results obtained by Kim and Dietmeyer using their symmetric function decomposition algorithm [51]. These counts too are for realizations before any technology mapping is done.

The last four columns provide literal counts for the multilevel realizations of the circuits after technology mapping. This mapping is performed to provide data that could be more fairly compared with the results generated by Decomp.

The "MIS 2" column reports values after mapping the author's MIS generated multilevel realizations to the same set of two-input gates that Decomp uses. The "MIS n" column reports the values after mapping onto a library of cells corresponding to those in the Oasis technology library. The "decomp" column gives values for the two-input-gate realization and the "decomp n" column gives values after mapping the two-input-gate realization onto the Oasis technology library.

Using the mapping data for comparison, one can conclude that Decomp outperforms MIS for symmetric functions. Not only are the circuits generated by Decomp

substantially smaller than those generated by MIS, Decomp requires much less time to produce a solution than does MIS.

A comparison between Decomp and the results of Kim and Dietmeyer is more difficult since there is no common unit of measure. However, if the mapping procedure gives an increase in literal count similar to that observed when mapping the MIS values, then Decomp generates smaller realizations than do Kim and Dietmeyer.

5.5.2 Arithmetic Functions

Table 5.5, compares Decomp's results with those of Oasis [15, 56] for a set of arithmetic functions.

Table 5.5: Comparison of Arithmetic Functions with Oasis
(a) Oasis, Decomp, Mcmap
(b) Decomp-r and Recomp technology mapping
*** - could not be decomposed by Decomp

	Decaf					Decomp					Mcmap				
func	gt	lit	trn	lvl	sec	gt	lit	trn	lvl	sec	gt	lit	trn	lvl	sec
adr4	27	61	118	7	4.9	17	48	102	7	1.1	17	48	82	6	5.3
adr5	38	87	170	7	13.0	22	62	132	9	59.9	25	65	112	8	5.4
add6	82	176	348	9	12.1	27	76	162	11	46.0	33	82	142	10	5.6
mdiv7	412	929	1858	16	21.3	***	***	***	***	***	***	***	***	***	***
sqr4	14	29	58	4	4.0	17	36	86	5	0.1	20	39	76	5	4.9
sqr6	87	202	404	10	5.4	***	***	***	***	***	***	***	***	***	***
mlp4	245	551	1100		10.4	***	***	***	***	***	***	***	***	***	***

(a)

	Decomp-r					Recomp				
func	gt	lit	trn	lvl	sec	gt	lit	trn	lvl	sec
adr4	17	48	84	7	1.1	18	49	84	7	3.2
adr5	22	62	108	9	59.4	23	63	110	8	3.0
add6	27	76	132	11	46.6	26	75	130	11	3.4
mdiv7	***	***	***	***	***	***	***	***	***	***
sqr4	16	35	70	4	0.1	17	36	66	5	3.1
sqr6	***	***	***	***	***	***	***	***	***	***
mlp4	***	***	***	***	***	***	***	***	***	***

(b)

Results similar to those for symmetric functions are observed for most of the arithmetic functions in Table 5.5. There are a couple of points to note however. One is that three of the arithmetic functions could not be decomposed by Decomp. This occurs because an image function is generated that is not two-place decomposable. This in turn is due to the fact that these functions do not exhibit much symmetry.

Another observation is that Decomp takes longer to realize several of the functions. This is due to the large number of cubes in the cube lists for those functions.

5.5.3 Control Functions

Table 5.6 compares Decomp's results with those of Oasis [15, 56] for a set of control functions.

Decomp does not perform as well on the control functions. Generally, the realizations generated by Decomp are significantly larger than those generated by Decaf. The results are improved somewhat by technology mapping but they are still worse than those of Decaf. Again Decomp is unable to decompose several of the control functions due to a generated image function that is not decomposable.

5.6 Summary

5.6.1 Advantages

There are several benefits to using the restricted class of two-place decompositions. First, avoiding the more complex general problem leads to efficient techniques for detecting, selecting and applying decompositions. This makes possible the generation of good multi-level realizations for systems of switching functions in reasonable computation time.

Second, the decompositions lead to a circuit that is expressed in its simplest form, composed of one and two-input gates. Such a circuit can be mapped directly onto

Table 5.6: Comparison of Control Functions with Oasis
(a) Oasis, Decomp, Mcmap
(b) Decomp-r and Recomp technology mapping
*** - could not be decomposed by Decomp

	Decaf					Decomp					Mcmap				
func	gt	lit	trn	lvl	sec	gt	lit	trn	lvl	sec	gt	lit	trn	lvl	sec
con1	15	32	64	6	4.0	39	69	182	10	0.2	23	55	104	7	5.4
f2	24	40	72	5	4.1	13	24	70	5	0.1	21	38	76	5	5.2
misex1	43	96	192	7	4.5	89	153	434	22	1.1	60	142	270	13	6.1
alu1	26	56	112	4	4.2	61	98	270	6	27.9	34	71	142	4	5.5
dc1	31	65	130	6	4.2	43	84	202	8	0.2	30	62	134	6	5.4
dk17	53	110	218	9	4.9	116	210	560	13	58.0	82	175	338	11	6.5
risc	81	141	256	7	5.3	107	194	538	11	5.0	91	181	352	9	5.6
wim	18	39	76	6	4.0	29	49	122	6	0.2	21	41	80	5	5.1
sao2	109	250	500	11	7.4	***	***	***	***	***	***	***	***	***	***

(a)

	Decomp-r					Recomp				
func	gt	lit	trn	lvl	sec	gt	lit	trn	lvl	sec
con1	30	60	128	8	0.2	20	55	104	7	3.4
f2	9	20	48	4	0.1	16	33	70	4	3.4
misex1	56	136	274	13	1.0	46	149	282	11	5.4
alu1	36	73	168	4	28.3	27	85	170	4	4.6
dc1	29	70	134	7	0.2	27	76	140	7	3.9
dk17	83	177	374	12	58.0	64	159	308	10	5.5
risc	85	172	420	9	5.0	79		358	8	5.2
wim	20	40	82	5	0.2	21	41	78	4	5.1
sao2	***	***	***	***	***	***	***	***	***	***

(b)

fine grain gate array structures such as Motorola's MPA1000 FPGA family [71] and Xilinx's XC8100 FPGA family [99]. Alternatively, various portions of the circuit may be combined into larger logic elements that are more suitable for a different structure.

Third, the developed procedure is quite good at dealing with functions that are incompletely specified. This is important not only because the procedure generates image functions that contain don't cares, but also because many "practical" real-life circuits are incompletely specified.

Additionally, since symmetry detection is at the heart of the two-place decomposi-

tion procedure, symmetric and partially symmetric functions are handled quite well. In practice designers tend to choose functions that exhibit some degree of symmetry. Thus two-place decomposition techniques lend themselves well to the synthesis of many practical circuits.

5.6.2 Disadvantages

Unfortunately, functions that contain very little or no symmetry are not well disposed to two-place decomposition techniques. This can be seen in Tables 5.4, 5.5, and 5.6. In fact functions that exhibit no two-variable symmetries are not two-place decomposable. When this situation arises, alternate approaches must be taken to arrive at a multi-level realization.

One of these approaches is to use a broader class of decompositions, but this results in increased algorithm complexity and more difficulty in determining a "good" realization for the function.

Another approach is to use a different multi-level synthesis method such as factoring. This may be a good approach, since the factoring methods tend to produce inferior results for symmetric functions but superior results for non-symmetric functions and for functions exhibiting little symmetry.

A third approach is to try to convert the function that does not contain any decompositions into one that does. A method which attempts to accomplish this is described in Chapter 6.

The choice of decomposition and the choice of functions for h_1 and h_2 have an impact on the existence of symmetries in future steps. Therefore the decomposition that is selected and the assignments that are made to the h_i at each step should be the *best* ones to achieve the most efficient multi-level circuit. Although the above described relatively simple decomposition selection heuristics produce favourable results, they

do fail in several cases.

Chapter 6 discusses several enhancements to the two-place decomposition procedure which exploit the information contained in the function's autocorrelation values to transform a function exhibiting no decompositions into one that does. In Chapter 7 an alternate technique is presented to decompose functions that exhibit no symmetry.

5.6.3 Conclusion

The enhanced two-place decomposition procedure has the capability of allowing the user to select between mappings for a CDNES decomposition. As shown, the use of different CDNES mappings can lead to considerable reductions in switching circuit size, both in terms of numbers of gates and gating levels.

Also the new decomposition merge procedure performs better than the original two-place decomposition procedure. It is also more flexible as it is readily extendible to allow selection of alternate mappings for other decomposition types. The procedure gives Decomp the ability to select particular mappings based on the existence of other mapping functions. For example, Decomp makes good CDNES decomposition mapping choices for most of the functions in the tables. In particular the choice of the AND/XOR mapping leads to the excellent results given in Tables 5.3 and 5.4.

As the tables show, for several of the functions Decomp out-performs Oasis in both speed and size of circuit realization. Decomp performs extremely well for symmetric functions and for functions with a large amount of symmetry such as the adder functions. However, as the cube lists grow in size, Decomp tends to be slower. Still, in most cases, Decomp is fast enough that one can try multiple runs with different mapping choices etc. in much the same way that multiple runs of MIS must often be made to tune its performance.

Chapter 6

Extending Two-Place Decomposition

It is possible for the initial function or for any image function that is generated in the decomposition process to exhibit no two-place symmetries, and therefore not be two-place decomposable. Two alternate paths can be followed to remedy this situation. First, a different multi-level synthesis procedure may be used to realize the function. Second, one can attempt to transform the function into one that does exhibit some symmetry.

This chapter describes work that follows the latter alternative and uses information contained in the total autocorrelation of a system of functions to perform transformations. While similar approaches have been previously implemented, this synthesis system is the first attempt of which the author is aware to combine them in a robust package applicable to systems of switching functions of significant size.

The chapter begins with Section 6.1, which details an algorithm for linearizing a system of switching functions. Section 6.2 analyses the effects of linearization on the size of a switching circuit realizing a switching function. Certain benchmark functions

are synthesized using various synthesis tools to determine these effects. Both two-level and multi-level circuits are generated for each function for the purposes of the analysis.

Next, the integration of linearization and two-place decomposition is considered. The results produced by the integrated system are compared with those produced by MIS [14]. These results indicate that the new system has considerable promise and is certainly worth developing further.

Section 6.3 identifies relationships between autocorrelation coefficients and the decompositions exhibited by a completely specified switching function. This leads to a linearization-like process that may be performed to enhance the performance of the two-place decomposition procedure.

Since the two-place decomposition procedure generates image functions that are incompletely specified, the autocorrelation-based procedures must manipulate don't cares. Section 6.4 describes a method that may be used for this purpose. The relationships identified in Section 6.3 are extended to incompletely specified functions.

Since not all switching functions can be made to be two-place decomposable, there are cases for which the two-place decomposition procedure fails to produce a realization. The chapter concludes with a discussion of alternatives to consider in this situation.

6.1 Linearization Procedure

As shown in Chapter 2, the first-order autocorrelation coefficients of a switching function may be used to compute the complexity estimate $C(f)$ of that switching function. Linearization of **B** [48] maximizes $C(f)$. The linearization procedure for a function yields a realization consisting of a linear prefilter, implemented as an n-input n-output XOR network, and a linearized function that is typically simpler than

the original in terms of the number of product terms required in a minimal sum-of-products expression.

Furthermore, since linearization corresponds to the spectral classification procedure, the linearized function is the canonical function for a class of functions. If the original function belongs to a class which contains a function that exhibits two-place symmetries, then this canonical function has or may be transformed to have two-place symmetries. Thus linearization frequently has the added benefit of transforming a switching function that exhibits no decompositions into one that does.

6.1.1 Linearization Algorithm

Algorithm 6.1 is an algorithm for linearizing a system of switching functions of n variables using the total autocorrelation of the system. It is based on the procedure described in [48], in which Karpovsky [48] has shown that the necessary transformation must be formed from a set of subscripts whose binary representations form a basis of the input binary vector space.

The coefficient with the largest value is selected so that the first-order coefficients are maximized. If there is more than one coefficient with the largest value. then the one with the lowest order is selected to minimize the number of prefilter XORs. The selection of the lexicographically last coefficient is an arbitrary choice.

At the end of the procedure the rows of L^{-1} identify the linear prefilter. Specifically row i of L^{-1} defines the XOR function that replaces x_{n-i+1}. The variables involved in the XOR function are those x_j for which column $n-j+1$ of row i is 1. L^{-1} is used to rearrange the truth table of the original system of functions to find the truth table of the linearized system of functions.

Algorithm 6.1: Linearization using Autocorrelation

Initialize L to the empty matrix

For $c = 1$ to n do

{

Select an autocorrelation coefficient other than b_0 as follows:

1) Select a coefficient with the largest value

2) within 1) select the coefficient of lowest order

3) within 2) select the lexicographically last coefficient

Add the binary representation of the decimal subscript of the selected coefficient as column c of L with the bit corresponding to x_i in row $n - i + 1$

Eliminate every autocorrelation coefficient with a binary representation of its decimal subscript that is the bit-by-bit mod 2 sum of some subset of the columns of L

}

Compute L^{-1}, the inverse mod 2 of L

6.1.2 Example Linearization

In Table 6.1 a four-variable function definition is provided along with its autocorrelation. Initially L is set to the empty matrix. The largest value coefficients, ignoring b_0, are b_7, b_{10}, and b_{13}. So b_{10} is selected first since it is the largest, lowest order, lexicographically last coefficient. Place the binary representation of 10 as the first column of L. Hence

$$L = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}.$$

Eliminate b_{10}, which leaves b_7 and b_{13} as the largest coefficients. They are both of the same order, so choose b_{13} since it is the lexicographically last of the two. Add 13 as a column to L giving

Table 6.1: Function for Example Linearization

#	$x_4\ x_3\ x_2\ x_1$	$f(X)$	B	#	$x_4\ x_3\ x_2\ x_1$	$f(X)$	B
0	0 0 0 0	0	6	8	1 0 0 0	1	2
1	0 0 0 1	0	0	9	1 0 0 1	0	2
2	0 0 1 0	1	2	10	1 0 1 0	0	4
3	0 0 1 1	0	2	11	1 0 1 1	1	0
4	0 1 0 0	0	2	12	1 1 0 0	0	2
5	0 1 0 1	1	2	13	1 1 0 1	0	4
6	0 1 1 0	0	0	14	1 1 1 0	0	2
7	0 1 1 1	1	4	15	1 1 1 1	1	2

$$L = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Remove coefficient b_{13}. Also remove b_7 since $0111 = 1010 \oplus 1101$. Of the remaining coefficients, b_8 is the largest, lowest-order, lexicographically last one. Choosing it yields

$$L = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Remove coefficients b_8, and b_2, b_5, and b_{15}. The final coefficient chosen is b_4, the largest, lowest-order, lexicographically last remaining coefficient. Add 4 to L giving

$$L = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

Then computing the mod 2 inverse of L yields

$$L^{-1} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}.$$

Thus the linear prefilter is:

$$\begin{aligned} x_1 &\leftarrow x_1 \oplus x_3, \\ x_2 &\leftarrow x_1 \oplus x_2 \oplus x_4, \\ x_3 &\leftarrow x_1, \\ x_4 &\leftarrow x_2. \end{aligned}$$

To obtain the truth table for the linearized function, perform a matrix multiplication, with all operations being done mod 2, between L^{-1} and the matrix of binary representations of the minterm numbers. The minterm numbers are placed into the columns of the matrix with the bit corresponding to x_i in row $n - i + 1$. Thus

$$\begin{bmatrix} 0010 \\ 0001 \\ 1011 \\ 0101 \end{bmatrix} \begin{bmatrix} 0000000011111111 \\ 0000111100001111 \\ 0011001100110011 \\ 0101010101010101 \end{bmatrix} = \begin{bmatrix} 0011001100110011 \\ 0101010101010101 \\ 0110011010011001 \\ 0101101001011010 \end{bmatrix}.$$

The linearized function is given in Table 6.2.

Table 6.2: Linearized Function Truth Table

$x_4\ x_3\ x_2\ x_1$	$f(X)$	$x_4\ x_3\ x_2\ x_1$	$f(X)$
0 0 0 0	0	1 0 0 0	0
0 0 0 1	0	1 0 0 1	0
0 0 1 0	1	1 0 1 0	1
0 0 1 1	0	1 0 1 1	0
0 1 0 0	0	1 1 0 0	1
0 1 0 1	0	1 1 0 1	0
0 1 1 0	1	1 1 1 0	1
0 1 1 1	0	1 1 1 1	1

6.2 Linearization Performance Analysis

This section examines the effects that linearization has on the complexity of the realizations of switching functions. As stated, linearization maximizes $C(f)$ by maximizing minterm adjacencies. However, the linearization process does not consider

the complexity of the prefilter required to perform the transformation. Thus any decrease in complexity of the realization of a function may be lost due to a high prefilter complexity.

Furthermore, maximizing minterm adjacencies has a greater impact on the complexity of two-level realizations of functions than on the complexity of multi-level realizations. The following section exemplifies this situation.

6.2.1 Pre-Linearization

This section provides the results of experiments that use several benchmark functions to illustrate the effects of linearization. Two realizations are generated for each benchmark function using each of three different synthesis systems. One realization is for the original function and the other is for the function after it is linearized. The synthesis systems are Espresso [13], MIS [14], and Decomp.

MIS is used for comparison purposes because it is a system that uses factoring techniques to synthesize systems of functions and a comparison between two multi-level synthesis techniques is useful. Also, since MIS is factoring-based and uses as a starting point a sum-of-products expression, it is useful to determine if any simplification that is achieved for two-level realizations also appears in the multi-level realization.

Tables 6.3 to 6.6 provide the results of the experiments. The tables contain two columns for each synthesis system to report the sizes of the realizations for the non-linearized and linearized functions respectively. The tables provide for each benchmark function:

- The number of product terms in the minimal two-level SOP forms as determined by Espresso.

- The number of two-input gates and the number of gating levels in each realization generated by MIS. The realization for the linearized function includes the prefilter.

- The numbers of two-input gates and gating levels in each realization generated by Decomp. The realization for the linearized function includes the prefilter. A "—" in the table indicates that Decomp could not realize the function because the function or a subsequent image function exhibited no decompositions.

- The number of two-input XOR gates and the number of gating levels required for the linear prefilter.

Table 6.3 reports the results for a set of single-output functions. The set consists of 18 four-variable functions, each being the representative function from one of the 222 NPN equivalence classes of four-variable functions presented by Harrison [34]. This particular set is the set of four-variable NPN-class representative functions identified by Miller [58] as exhibiting no two-place decompositions.

As the table shows, the original functions indeed have no decompositions. However, after being linearized every one exhibits decompositions and a complete realization is found for it. The realizations generated by Decomp for the linearized functions are equal to or somewhat larger than those generated by MIS except for *f2*.

For MIS, in all but one case there is a reduction in the size of the realizations of the linearized functions from the realizations of the original functions. This reduction corresponds to the reduction observed for the two-level realizations generated by Espresso. For four of the functions however, there is an increase in the number of gating levels.

Table 6.4 provides results for a set of symmetric functions. There is a large difference in the number of product terms required in PLA realizations of the original

Table 6.3: Linearization of 18 Four-variable NPN Representative Functions
— - could not find a decomposition

	Espresso		MIS				Decomp				Filter	
			Orig		Linear		Orig		Linear			
Func	Orig	Lin	Gt	Lv	Gt	Lv	Gt	Lv	Gt	Lv	Gt	Lv
f1	4	2	15	6	8	6	——		9	6	3	2
f2	4	3	12	6	10	5	——		9	5	2	1
f3	4	3	15	6	9	6	——		11	6	3	2
f4	4	3	12	6	8	5	——		11	6	2	1
f5	4	3	13	5	10	6	——		11	5	3	1
f6	4	2	13	6	5	3	——		5	3	2	1
f7	4	2	14	6	7	5	——		7	5	3	2
f8	5	2	15	6	6	5	——		6	5	3	3
f9	4	2	15	6	8	5	——		8	5	4	2
f10	5	2	13	6	9	6	——		9	6	5	3
f11	4	2	13	5	8	5	——		9	6	3	1
f12	4	3	9	5	7	5	——		10	5	2	1
f13	4	3	11	5	8	5	——		11	6	3	1
f14	5	3	15	6	8	5	——		8	5	3	2
f15	5	3	11	5	8	6	——		11	6	3	2
f16	3	3	7	4	8	6	——		13	7	1	1
f17	5	3	15	6	9	6	——		11	7	3	2
f18	6	3	13	5	7	6	——		7	5	3	3

and linearized functions. The linearized functions require, on average, 51% fewer product terms.

However, the effects on the multi-level realizations are not as pronounced. For the realizations generated by MIS, the required number of two-input gates decreases a little and even increases for *rd73*. For three, or 50% of the functions, the decrease in the number of gates is accompanied by an increase in the number of levels.

Decomp generates larger realizations for the linearized functions than for the non-linearized functions. Thus for Decomp, pre-linearization has a negative effect for symmetric functions. In either case however, Decomp generates much more compact realizations for these functions than does MIS with respect to both the number of

Table 6.4: Linearization Effects on Symmetric Function Realizations

			MIS					
	Espresso		Orig	Linear	Orig	Linear	Filter	
Func	Orig	Lin	Gt Lv	Gt Lv	Gt Lv	Gt Lv	Gt Lv	
2of5	10	6	23–8	18–10	13–6	14–7	4–3	
9sym	85	56	68–13	58–13	30–8	33–8	8–1	
rd53	31	12	29–10	28–11	16–6	22–9	9–4	
rd73	127	78	71–20	89–16	22–6	36–11	6–3	
rd84	255	156	144–17	132–18	28–6	44–11	7–3	
xor5	16	1	7–6	4–3	4–3	4–3	4–3	

gates and the number of gating levels.

Table 6.5 lists the results for a set of arithmetic benchmark functions. This table highlights a few important observations. First, and most obvious, is that Decomp, using two-place decomposition alone, is unable to generate complete realizations for the majority of the functions. For those functions that Decomp can synthesize, it generates realizations with fewer gates and at most the same number of gating levels as the MIS generated realizations.

Again, all of the Decomp results for linearized functions are larger than for the original functions. However, even the larger linearized results from Decomp are generally better than those generated by MIS for the original, non-linearized functions.

Another observation is, that except for the adder functions, which exhibit a large degree of symmetry and XOR structure, the differences between the Espresso results for the original and linearized functions are not as dramatic as they are for the symmetric functions.

For MIS, fewer of the linearized function realizations are smaller and the reduction in size is marginal or is offset by an increase in the number of gating levels. Also, for 38% (9 out of 24) of the functions listed in Table 6.5, the linearized-function realizations are larger than the original-function realizations. This is more than twice

Table 6.5: Linearization Effects on Arithmetic Function Realizations
— - could not find a decomposition

			MIS				
	Espresso		Orig	Linear	Decomp Orig	Decomp Linear	Filter
Func	Orig	Lin	Gt | Lv	Gt | Lv	Gt | Lv	Gt | Lv	Gt | Lv
add6	355	37	68–20	55–19	27–11	59–20	11–2
adr2	11	5	10–5	12–8	7–3	11–6	3–2
adr3	31	10	24–9	21–10	12–5	20–8	5–2
adr4	75	17	35–13	33–15	17–7	33–12	7–2
adr5	167	26	42–16	44–17	22–9	46–16	9–2
dist3	36	31	86–13	107–17	——	——	6–4
dist4	120	105	283–44	261–41	——	——	9–2
fl6	40	31	106–29	104–20	——	——	7–4
fl8	129	124	369–55	380–59	——	——	9–3
mdiv7	203	111	384–17	363–18	——	——	8–3
mlp2	7	7	9–3	9–3	8–3	8–3	0–0
mlp3	31	32	77–12	73–10	——	——	0–0
mlp4	124	123	311–46	303–50	——	——	0–0
root6	23	20	53–10	58–12	——	——	7–2
root7	34	28	82–14	78–15	——	——	6–3
root8	57	47	129–20	146–22	——	——	11–3
root9	85	75	203–17	188–36	——	——	12–3
sqr2	3	3	3–2	3–2	3–2	3–2	0–0
sqr3	7	7	11–4	11–4	7–4	7–4	0–0
sqr4	13	11	22–5	19–5	17–5	19–6	1–1
sqr5	26	27	54–8	56–11	——	——	1–1
sqr6	48	48	126–26	136–12	——	——	0–0
sqr8	180	184	472–61	481–54	——	——	0–0
z4ml	59	17	35–14	27–12	15–7	27–13	7–3

the 17% seen in Table 6.4.

Since heuristics are used in each of the synthesis systems, the results vary, even with the order in which the inputs are specified. This is illustrated for Espresso and MIS by the functions *mlp3*, *mlp4*, and *sqr8*, for which, the linearization process generates no prefilter and only reorders the inputs. For these "linearized" functions, Espresso generates larger realizations for *mlp3* and *sqr8* and a smaller one for *mlp4*. MIS, produces smaller realizations for *mlp3* and *mlp4* and a larger one for *sqr8*.

Table 6.6: Linearization Effects on Control Function Realizations
— - could not find a decomposition

	Espresso		MIS				Decomp				Filter	
			Orig		Linear		Orig		Linear			
Func	Orig	Lin	Gt	Lv	Gt	Lv	Gt	Lv	Gt	Lv	Gt	Lv
postal	18	18	36	9	30	10	——		32	4	5	1
5xp1	64	68	106	12	120	13	——		——		3	1
bw	22	22	216	8	195	11	——		——		3	3
con1	9	11	23	6	28	10	39	10	47	10	3	2
f2	8	8	24	7	24	7	13	5	13	5	0	0
f51m	76	70	100	14	113	16	——		——		10	2
misex1	12	12	66	10	57	11	86	22	71	16	4	2
sao2	58	45	165	13	146	17	——		——		11	4
alu1	19	19	44	5	44	5	61	6	77	7	0	0
alu2	68	68	87	15	87	11	——		——		0	0
alu3	65	65	82	14	86	10	——		——		0	0
apla	25	25	121	9	132	10	——		——		1	1
clip	118	49	146	17	133	15	——		——		9	3
dc1	9	9	44	8	42	10	43	8	44	10	3	3
dc2	39	37	111	13	131	14	——		——		6	2
dk17	18	18	75	10	77	9	116	13	119	16	2	1
dk27	10	10	22	6	22	10	25	7	25	7	0	0
risc	28	28	98	8	102	8	99	11	87	9	2	1
sqn	38	29	107	12	73	13	——		——		5	2
wim	9	9	24	6	22	7	29	6	30	8	2	2

Table 6.6 shows the results for a set of control functions. Except for *clip*, there is very little difference between the sizes of non-linearized and linearized function realizations generated by Espresso. Again the realizations generated by Decomp for the linearized functions are generally larger than for the original functions. 35% (7 of 20) of the MIS results suffer from the same problem.

Function *alu1* shows how the heuristics in Decomp are affected by variable order. For this function, linearization produces a permutation of the inputs, which causes a different order of decomposition selection by the two-place decomposition process, that obviously leads to a different, and worse result.

Function *postal* is noteworthy for two reasons. First it is the largest function (eight variable) yet found that can be linearized to have a two-place decomposition. Second, this function is used in [58] as an example of a complex design problem in which two-place decomposition is used as a tool to aid in the design process. This function checks the validity of either the first two-character or last two-character component of the Canadian postal code.

In [58] a rather lengthy manual analysis of the function is performed to redefine the problem using a set of decomposable functions which are combined, again manually, to form the final output. The realization that results contains 71 gates, a significant portion of which is generated by hand to combine the several functions. Two-place decomposition, with the aid of linearization, is able to synthesize *postal* in 32 gates with a five-gate prefilter and without manual intervention.

6.2.2 Linearization to Remedy Lack of Decompositions

As seen in the previous section the two-place decomposition procedure is unable to generate realizations for over half of the benchmark functions considered. However, the data in the tables provide the results for when the synthesis process is abandoned when no decompositions are found. If instead, linearization is used during the decomposition process, and not used just as a step to be performed prior to synthesizing a realization, a larger number of the functions can be realized using two-place decomposition.

At any point the two-place decomposition process may generate an image function that exhibits no symmetry. When this occurs Decomp performs a linearization on the image function in an attempt to transform it into one that does have symmetries. If the linearization process fails to yield an image function that can be decomposed, then Decomp abandons the synthesis process.

Table 6.7: Decomp Results using Linearization during Decomposition
— - could not find a decomposition

	Original				Linear			
	Circuit		Filters		Circuit		Filters	
Func	Gt	Lv	Gt	Lv	Steps	Gt	Lv	Gt	Lv	Steps
5xp1	146	25	25	7	4	129	21	19	7	3
bw	233	12	3	3	1	264	13	13	6	2
f51m	——		——		—	142	27	29	8	3
mdiv7	150	16	3	2	1	——		——		—
root7	106	28	17	6	3	——		——		—
root9	813	134	287	62	18	——		——		—
sqr5	55	10	3	1	1	54	11	4	2	2
sqr6	196	31	49	13	6	166	28	36	12	6
alu3	——		——		—	138	20	12	4	2
apla	204	26	10	3	1	205	21	11	5	3
clip	128	22	9	4	2	132	19	17	7	2
dc2	246	43	57	15	6	210	36	34	13	6
sqn	90	30	29	12	6	——		——		—

Table 6.7 provides results for multi-level realizations generated by Decomp using linearization to remedy a lack of decomposition. Both original and pre-linearized functions are used. For the original functions and the pre-linearized functions, the table provides the total number of two-input gates and the number of gating levels for the multi-level realization, the total number of filter gates and gating levels used, and the number of linearization steps performed.

As the data show, linearization can be used to transform a system of functions that exhibits no decomposition into one that does. Use of this technique enables the two-place decomposition process to generate realizations for some functions that it otherwise would fail to synthesize.

Almost all of the realizations generated for these functions by Decomp are larger than the realizations generated by MIS. In only three cases, for *mdiv7*, *clip* and *sqn*, did Decomp generate better results than MIS. For the others the Decomp results range from one gate worse (*sqr5*) to 400% worse (*root9*).

6.2.3 Linearization Summary

This section examines the use of linearization for two purposes. First, it is used as a step prior to synthesis to determine if linearization does indeed simplify the realizations of switching functions. The tables given in Section 6.2.1 provide the data from the analysis.

The tables indicate that linearization generally simplifies the two-level realizations of the systems of functions. The degree of simplification for a function's two-level realization is highly dependent on the function's structure. Functions that contain substantial XOR structure such as adders benefit greatly from linearization. This is not surprising considering that the prefilter consists of an XOR network. Symmetric functions also benefit well from linearization. However, functions that are more random in definition are less affected.

Pre-linearization, except in a few cases, does not simplify multi-level realizations. This is observed in most of the Decomp results and in several cases for MIS. Although linearization tends to simplify two-level realizations, it doesn't have much benefit for multi-level realizations.

Indeed, in almost all cases, Decomp generates larger multi-level realizations for the linearized functions than for the original functions. Thus performing a pre-linearization step generally has no benefit for two-place decomposition, especially for those functions for which it performs extremely well, such as symmetric functions, and functions that exhibit significant amounts of symmetry, like the adders. For these functions, Decomp's realizations are on average 54% smaller than those generated by MIS.

On the other hand, two-place decomposition does not perform as well for functions that exhibit little symmetry. For these, the realizations from Decomp are on average 21% larger than those generated by MIS. For 50% of the functions. the two-place

decomposition process fails to generate realizations.

The second purpose for linearization is to use it during the two-place decomposition process to attempt to transform a non-decomposable image function into a decomposable one. When this is done two-place decomposition succeeds in several more cases, but again the results are generally not as good as those produced by MIS. For these functions Decomp's results are on average 50% larger. If the *root9* result, which is the only vastly different result, is omitted from this analysis then Decomp's results are about 29% larger than MIS's.

From these results, it is clear that the reasons why pre-linearization causes larger multi-level realizations need to be examined. Additionally, to perform the linearization steps for the functions in Table 6.7, Decomp has to handle don't cares. For this purpose, Decomp uses the $\{+1, 0, -1\}$ encoding for the image functions, which assigns the value 0 to don't cares and thus ignores them. Although this method is successful in transforming some of the image functions to have decompositions, it does not take advantage of the don't cares. A better approach, based on the relationship between autocorrelation values and the decompositions that a function exhibits, is developed in the following sections.

6.3 Autocorrelation-Decomposition Relationship

Linearization maximizes adjacency, but that is not a requirement for a two-place decomposition to exist. The following explores the autocorrelation characterization associated with two-place decomposition to gain insight into possible transformations that may be performed to simplify a function with respect to two-place decomposition.

The following theorems establish relationships between the values of some autocorrelation coefficients and the decompositions that a completely specified switching function exhibits. For each of the proofs the truth vector $\mathbf{Z}$ of the switching

function is partitioned into a 2^{n-2} row by 4 column matrix with columns labeled $k, k = 0, \ldots, 3$, corresponding to the possible assignments to a variable pair (x_i, x_j). $\mathbf{Z}^{k_l}$, $k_l \in \{0, 1, 2, 3\}$, $l = 1, \ldots, 4$ are the columns of the matrix. Additionally, the function

$$\alpha(\mathbf{Z}^i, \mathbf{Z}^j) = 2 \sum_{k=0}^{2^{n-2}-1} \mathbf{Z}_k^i \mathbf{Z}_k^j,$$

where $\mathbf{Z}_k^i$ is the kth element of column vector $\mathbf{Z}^i$, computes the contribution made to an autocorrelation coefficient by the two columns $\mathbf{Z}^i$ and $\mathbf{Z}^j$.

Theorem 6.1 *If a completely specified function $f(X), X = \{x_1, \ldots, x_n\}$ exhibits a non-exclusive-OR simple-disjunctive two-place decomposition in variables x_i and x_j, then $b_u = b_v = b_{u+v}$ where $\|u\| = \|v\| = 1$, $u_i = 1$, $v_j = 1$.*

Proof: If $f(X)$ has a non-exclusive-OR simple-disjunctive decomposition, then

$$\mathbf{Z}^{k_1} = \mathbf{Z}^{k_2} = \mathbf{Z}^{k_3}, \mathbf{Z}^{k_1} \neq \mathbf{Z}^{k_4},$$

for distinct $k_l \in \{0, 1, 2, 3\}$. Then $\alpha(\mathbf{Z}^{k_1}, \mathbf{Z}^{k_2}) = \alpha(\mathbf{Z}^{k_1}, \mathbf{Z}^{k_3}) = \alpha(\mathbf{Z}^{k_2}, \mathbf{Z}^{k_3})$. Assign this value to E_0. It follows that $\alpha(\mathbf{Z}^{k_1}, \mathbf{Z}^{k_4}) = \alpha(\mathbf{Z}^{k_2}, \mathbf{Z}^{k_4}) = \alpha(\mathbf{Z}^{k_3}, \mathbf{Z}^{k_4})$. Assign this value to E_1.

From equation 2.9:

$$\begin{aligned} b_{w_1} &= \alpha(\mathbf{Z}^{k_1}, \mathbf{Z}^{k_3}) + \alpha(\mathbf{Z}^{k_2}, \mathbf{Z}^{k_4}) = E_0 + E_1, \\ b_{w_2} &= \alpha(\mathbf{Z}^{k_1}, \mathbf{Z}^{k_2}) + \alpha(\mathbf{Z}^{k_3}, \mathbf{Z}^{k_4}) = E_0 + E_1, \\ b_{w_3} &= \alpha(\mathbf{Z}^{k_1}, \mathbf{Z}^{k_4}) + \alpha(\mathbf{Z}^{k_2}, \mathbf{Z}^{k_3}) = E_1 + E_0, \end{aligned}$$

for distinct $w_c \in \{u, v, u+v\}$, $c = 1, 2, 3$. Therefore $b_u = b_v = b_{u+v}$. □

Theorem 6.2 *A completely specified switching function $f(X), X = \{x_1, \ldots, x_n\}$ exhibits a simple-disjunctive two-place XOR decomposition in variables x_i and x_j if, and only if, $b_{u+v} = b_0$, where $\|u\| = \|v\| = 1$, $u_i = 1$, $v_j = 1$.*

Proof: If $f(X)$ has a simple-disjunctive XOR decomposition in variables x_i and x_j, then $00 \sim 11$ and $01 \sim 10$. Therefore, $\mathbf{Z}^0 = \mathbf{Z}^3$ and $\mathbf{Z}^1 = \mathbf{Z}^2$.

From equation 2.9

$$b_{u+v} = \alpha(\mathbf{Z}^0, \mathbf{Z}^3) + \alpha(\mathbf{Z}^1, \mathbf{Z}^2)$$

and

$$b_0 = \frac{1}{2}\alpha(Z^0, Z^0) + \frac{1}{2}\alpha(Z^1, Z^1) + \frac{1}{2}\alpha(Z^2, Z^2) + \frac{1}{2}\alpha(Z^3, Z^3).$$

Since $\mathbf{Z}^0 = \mathbf{Z}^3$ and $\mathbf{Z}^1 = \mathbf{Z}^2$ then

$$b_0 = \alpha(\mathbf{Z}^0, \mathbf{Z}^3) + \alpha(\mathbf{Z}^1, \mathbf{Z}^2).$$

Therefore, $b_{u+v} = b_0$.

If a second-order autocorrelation coefficient $b_{u+v} = b_0$, where $\|u\| = \|v\| = 1$, $u_i = 1$, $v_j = 1$, then $\mathbf{Z}^0 = \mathbf{Z}^3$ and $\mathbf{Z}^1 = \mathbf{Z}^2$ with respect to variables x_i and x_j must be true. Therefore $00 \sim 11$ and $01 \sim 10$ and hence $f(X)$ exhibits a simple-disjunctive two-place XOR decomposition in variables x_i and x_j. □

Theorem 6.1 identifies a necessary, but not a sufficient condition for a non-XOR simple disjunctive decomposition to exist in a switching function. Figure 6.1 provides a decomposition chart of an example four-variable function, $f(x_1, x_2, x_3, x_4)$, that exhibits no symmetries in variables x_1 and x_2 even though $b_1 = b_2 = b_3 = 2$.

Theorem 6.2 provides a necessary and sufficient condition for an XOR decomposition to exist in a switching function.

Theorem 6.3 *If a completely specified function $f(X), X = \{x_1, \ldots, x_n\}$ exhibits a simple-non-disjunctive two-place decomposition in variables x_i and x_j, then $b_{u+v} = b_u$ if the compatible assignments agree in x_i, or $b_{u+v} = b_v$ if the compatible assignments agree in x_j. $\|u\| = \|v\| = 1$, $u_i = 1$, $v_j = 1$.*

Figure 6.1: Example of Function with $b_u = b_v = b_{u+v}$ but no SD decomposition

x_4x_3 \ x_2x_1	00	01	10	11
00	0	0	0	1
01	0	0	1	1
10	0	1	0	1
11	1	0	0	1

Autocorrelation Coefficients

u	b_u	u	b_u
0	7	8	4
1	2	9	2
2	2	10	4
3	2	11	2
4	4	12	4
5	4	13	2
6	2	14	2
7	2	15	4

Proof: If $f(X)$ has a simple-disjunctive decomposition, then $\mathbf{Z}^{k_1} = \mathbf{Z}^{k_2}$ for some k_1 and k_2 at distance 1. So $\alpha(\mathbf{Z}^{k_1}, \mathbf{Z}^{k_3}) = \alpha(\mathbf{Z}^{k_2}, \mathbf{Z}^{k_3})$. Assign this value to E_0. Also $\alpha(\mathbf{Z}^{k_1}, \mathbf{Z}^{k_4}) = \alpha(\mathbf{Z}^{k_2}, \mathbf{Z}^{k_4})$ Assign this value to E_1. Let k_3 be at distance 2 from k_1 and k_4 be at distance 2 from k_2. Then

$$\begin{aligned} b_{u+v} &= \alpha(\mathbf{Z}^{k_1}, \mathbf{Z}^{k_3}) + \alpha(\mathbf{Z}^{k_2}, \mathbf{Z}^{k_4}) \\ &= E_0 + E_1. \end{aligned}$$

Assume without loss of generality that k_1 and k_2 agree with respect to variable x_i. (Agreement in variable x_j would simply be a relabeling). Then

$$\begin{aligned} b_u &= \alpha(\mathbf{Z}^{k_1}, \mathbf{Z}^{k_4}) + \alpha(\mathbf{Z}^{k_2}, \mathbf{Z}^{k_3}) \\ &= E_1 + E_0, \\ b_v &= \alpha(\mathbf{Z}^{k_1}, \mathbf{Z}^{k_2}) + \alpha(\mathbf{Z}^{k_3}, \mathbf{Z}^{k_4}). \end{aligned}$$

Thus $b_u = b_{u+v}$. □

Theorem 6.4 *If a completely specified function $f(X), X = \{x_1, \ldots, x_n\}$ exhibits a complex-disjunctive two-place decomposition in variables x_i and x_j, then $b_u = b_v$, where $\|u\| = \|v\| = 1$, $u_i = 1$, $v_j = 1$.*

Proof: If $f(X)$ has a complex-disjunctive decomposition, then $\mathbf{Z}^{k_1} = \mathbf{Z}^{k_2}$, for some k_1 and k_2 that are at distance 2. So $\alpha(\mathbf{Z}^{k_1}, \mathbf{Z}^{k_3}) = \alpha(\mathbf{Z}^{k_2}, \mathbf{Z}^{k_3})$. Assign this value to E_0. Also $\alpha(\mathbf{Z}^{k_1}, \mathbf{Z}^{k_4}) = \alpha(\mathbf{Z}^{k_2}, \mathbf{Z}^{k_4})$ Assign this value to E_1.

Assume without loss of generality that k_3 is the remaining column that is at distance 1 from k_1 with respect to x_i and at distance 1 from k_2 with respect to x_j. Similarly k_4 is the remaining column that is at distance 1 from k_2 with respect to x_i and at distance 1 from k_1 with respect to x_j.

From equation 2.9:

$$\begin{aligned} b_u &= \alpha(\mathbf{Z}^{k_1}, \mathbf{Z}^{k_3}) + \alpha(\mathbf{Z}^{k_2}, \mathbf{Z}^{k_4}) = E_0 + E_1, \\ b_v &= \alpha(\mathbf{Z}^{k_1}, \mathbf{Z}^{k_4}) + \alpha(\mathbf{Z}^{k_2}, \mathbf{Z}^{k_3}) = E_1 + E_0. \end{aligned}$$

Therefore $b_u = b_v$. □

As for Theorem 6.1 the converses of Theorems 6.3 and 6.4 do not necessarily hold. This is seen for the function in Figure 6.1, where $b_4 = b_5$, but the function has no simple non-disjunctive decomposition in the corresponding variables x_1 and x_3. Similarly $b_4 = b_8$ and no complex disjunctive decomposition in x_3 and x_4 exists.

Theorem 6.5 is included here for completeness.

Theorem 6.5 *A variable x_i is redundant in a completely specified switching function $f(X), X = \{x_1, \ldots, x_n\}$ if, and only if, $b_u = b_0, \|u\| = 1, u_i = 1$.*

Proof: Partition $\mathbf{Z}$ into a 2^{n-1} by 2 column matrix, with columns $\mathbf{Z}^0$ and $\mathbf{Z}^1$ corresponding to the two possible assignments to x_i. If x_i is redundant, then $\mathbf{Z}^0 = \mathbf{Z}^1$ and

$$b_u = \alpha(\mathbf{Z}^0, \mathbf{Z}^1) = b_0.$$

If some first order coefficient, $b_u = b_0, \|u\| = 1, u_i = 1$, then $\mathbf{Z}^0 = \mathbf{Z}^1$. Therefore x_i is redundant since both possible assignments to x_i yield the same switching function values. □

Table 6.8: Autocorrelation Coefficient Value Relationships for Decompositions

Decomposition Type	Autocorrelation Value Relationships $\|\|u\|\| = \|\|v\|\| = 1, u \neq v$	
Simple Disjunctive (non-XOR)	$b_u = b_v = b_{u+v}$	(necessary)
Simple Disjunctive (XOR)	$b_{u+v} = b_0$	(necessary & sufficient)
Simple Non-disjunctive	$b_{u+v} = b_w, w \in \{u, v\}$	(necessary)
Complex Disjunctive	$b_u = b_v$	(necessary)

Table 6.8 provides a summary of the autocorrelation coefficient value relationships that exist for each decomposition type.

The above theorems may be used to perform a linearization-like procedure that is targeted to two-place decomposition. Theorem 6.2 leads to a possible linearization step, namely if some $b_u = b_0, u \neq 0, \|u\| > 2$, then that coefficient may be moved to a second order position using linear translation operations. (If $\|u\| = 1, u_i = 1$ then variable x_i is redundant, as shown in Theorem 6.5.) Once in a second order position the function is guaranteed to exhibit an SD XOR decomposition and one variable may be removed from the problem immediately.

In addition, heuristics based on Theorems 6.1, 6.3, and 6.4 may be applied. For example, if a function's autocorrelation contains no $b_u = b_0$ then the linearization may, instead of maximizing the values of the first order coefficients, make a corresponding set of first and second-order coefficients equal, which corresponds to a non-XOR simple-disjunctive decomposition.

However, Figure 6.1 demonstrates how difficult it can be to predict what decompositions are exhibited by a function. At least one of the autocorrelation coefficient value relationships described in Theorems 6.1, 6.3, and 6.4 holds for every pair of variables. Yet, the function exhibits no decompositions in any of them.

6.4 Incompletely Specified Functions

The preceding section deals with completely specified functions. A major purpose of a process such as two-place decomposition is to introduce don't cares into a function to create an image function that is defined for fewer input conditions so that the image is "smaller". Also the don't cares tend to increase the number of symmetries that exist. Thus for functions that initially are completely specified, the two-place decomposition procedure eventually generates image functions that contain don't cares.

To be of value in two-place decomposition, spectral techniques must effectively and meaningfully manipulate incompletely specified functions. Some work in the area of symmetry detection in partially specified functions has been done in the spectral domain [39] but little, if any has been done in the area of linearization or other simplification techniques.

If $\{+1, 0, -1\}$ encoding is used, then the autocorrelation may be computed for incompletely specified functions. Here the don't cares can be interpreted as contributing the average value over all possible 0 and 1 assignments to the don't cares or they can be viewed as contributing nothing. This does not provide much information with respect to two-place decomposition. An alternate method is required.

Karpovsky [49] suggests the use of a *cross correlation* [48] of a pair of recoded truth vectors to describe an incompletely specified switching function $f(X)$: one with 1's where there are 0's in the original function, and one with 1's where there are 1's in the original function. Let $f^0(X)$ and $f^1(X)$ respectively, represent these two recoded vectors. The cross correlation of $f^0(X)$ and $f^1(X)$,

$$\mathrm{B}(u) = \sum_{v=0}^{2^n-1} f^0(v) f^1(v \oplus u), \tag{6.1}$$

is used to determine characteristics of the original incompletely-specified function.

With respect to two-place decomposition, the value of coefficient b_u of this cross-correlation provides the number of incompatible function values at distance $\|u\|$.

With respect to a decomposition chart labeled by variables $x_i, \forall i | u_i = 1$, this gives the number of places in which one column has a 0 entry and the corresponding entry in the corresponding column at distance $||u||$ has a 1 entry.

The cross-correlation may also be computed using the spectra of $f^0(X)$ and $f^1(X)$, denoted as $\mathbf{R}^0$ and $\mathbf{R}^1$ respectively, as

$$B(u) = \frac{1}{2^n}\mathbf{T}^n R^0 R^1. \tag{6.2}$$

To convert these coefficients to the number of compatible values of $f(X)$ at distance $||u||$ perform the computation

$$b_u \leftarrow 2^n - 2b_u,$$

for all u.

6.5 CDNES XOR Mapping Functions

As seen in Chapter 5 the choice of an XOR mapping function for one of the h_i of a CDNES decomposition can have a considerable impact on the size of the resulting circuit. What the use of the XOR function in a CDNES mapping stage represents is a linear translation operation discussed in Chapter 2, specifically it is the Type 4 translation listed under spectral invariance operations.

If a function has a CDNES decomposition in variables x_i and x_j then using the AND/XOR mapping functions is the equivalent of implementing $h_1 = x_i x_j$ and $h_2 = x_i + x_j$ followed immediately by a replacement of h_2 with $h_1 \oplus h_2$. Using the OR/XOR mapping functions is the equivalent of implementing $h_1 = x_i x_j$ and $h_2 = x_i + x_j$ followed immediately by a replacement of h_1 with $h_1 \oplus h_2$.

As seen in Chapter 2 this linear translation has the effect of interchanging 2^{n-2} pairs of coefficients in the spectral domain. In the function domain it has the effect of interchanging 2^{n-2} pairs of function values, i.e. it permutes $\mathbf{Z}$.

An excellent example of the impact of the AND/XOR mapping on a function that requires linearization for a complete realization to be generated is *mdiv7*, which needs 622 gates, 72 gating levels, and several linearization steps with AND/OR but needs only 150 gates, 16 levels and one linearization step when AND/XOR is used.

Therefore it is important to determine when the XOR should be used to help minimize the size of the function's realization. To do this effectively using the autocorrelation, don't cares must be handled. A method for this is suggested in section 6.4.

6.6 Conclusion

This chapter examines the effects of linearization on various types of circuit realizations and it concludes that pre-linearization doesn't actually help for two-place decomposition. In fact it makes things worse; hence, linearization should be used only when a function is not two-place decomposable. Also, for two-level realizations, linearization is more effective for symmetric functions and for partially symmetric functions that exhibit a high degree of symmetry than for non-symmetric functions.

Relationships between certain autocorrelation coefficient values and the decompositions that a function exhibits are identified. These relationships have not appeared in the literature. They suggest a new approach to the linearization of a function that targets two-place decomposition.

Some mappings that are available in the decomposition process, specifically those involving the XOR, are identified as a combination of a decomposition mapping immediately followed by a linear translation. Since some of these linear translation mappings make a significant difference in the size of a circuit, it would be beneficial to be able to identify when they should be used. Also a linear translation can be used independently of a complex-disjunctive decomposition mapping stage to maintain or create two-place decompositions in an image function.

An area for future work is to identify, using the autocorrelation of a switching function, if the use of a linear translation would be advantageous. It would be useful as well to be able to choose "good" mappings for all decomposition types using information other than the decompositions that have been found so far. Here too the autocorrelation may be of use.

Linearization is used to attempt to transform a non-decomposable function into a decomposable one. Currently, if this fails to yield a decomposable function, synthesis of the function is abandoned and a complete realization is not generated for it. In this case, other approaches need to be used to realize either, the original function in its entirety, or only the remaining image function.

One alternate approach is to divide the system of functions into a set of smaller subfunctions. Then the subfunctions would be re-composed at the end using a multiplexor. Since it is known that all four-variable functions are decomposable, possibly with a linearization step, one can always divide a function until it is composed of subfunctions of no more than four variables.

To do this effectively, the selection of variables to use as selectors to the recomposition multiplexors should be chosen such that as much circuitry as possible is shared among the subfunctions. This is a similar problem to that of choosing an optimal variable pairing for PLA optimization as discussed in Chapter 3. The theorems establishing relationships of autocorrelation coefficients to decompositions could well prove to be useful in this regard in that a measure of closeness to a decomposition would dictate which variable(s) to choose.

Chapter 7 presents a new approach that decomposes a large function into smaller subfunctions that two-place decomposition can handle. It performs fairly well on large functions and functions that are not two-place decomposable.

Currently Decomp does not perform a search of solutions for a "best" one. A future enhancement would be to have Decomp traverse several solution paths to determine

a good solution. Techniques limiting the search would have to be employed due to a possibly huge search space. The user could also interact with this search procedure or even control it. A similar approach is taken in MIS where the user specifies interactively (or through a command script) the procedures that MIS is to perform. This would make the whole process much more flexible than the batch-oriented process that currently takes place. Also the user can experiment with different parameters and options at various stages in the procedure instead of having to select a set of options at the beginning and have them be in effect throughout the entire procedure.

Also the user can guide the procedure instead of depending on a predefined set of heuristics and rules to guide the procedure. (Especially since one set of heuristics can not apply to all possible functions.)

Chapter 7

Binary Decision Diagrams

As mentioned earlier, a truth table may be used to represent a switching function. Other methods include the Karnaugh map [43] and partition matrices. However, since each of these methods lists the function's value at all 2^n possible input assignments, they quickly become unwieldy. A more compact representation is the cube list. Unfortunately, this representation too can be quite large, since a cube list could have one cube for every minterm, thereby containing 2^{n-1} elements in the totally-specified case (e.g. the XOR of n variables). This can rise to 2^n in the situation where 0's and 1's are explicitly identified with missing minterms taken as don't-cares. The latter is the scheme used in this dissertation.

Another construct for function representation that has recently gained popularity is the *Binary Decision Diagram* (*BDD*) [5, 55] and more specifically the *Reduced Ordered Binary Decision Diagram* (*ROBDD*) [17]. This chapter discusses BDDs and ROBDDs, describing their basic structure and operations relevant to the theme of this dissertation.

The primary advantage of BDDs is that the complexity of representing and manipulating switching functions can be substantially smaller than for other representations of switching functions. The same is true for the representation, computation

and manipulation of spectra and autocorrelation of switching functions.

This chapter shows that ROBDDs can be used effectively in representing and manipulating switching functions and their spectra and autocorrelations. ROBDDs are also quite useful for multi-level synthesis, and several ROBDD-based procedures have been developed [41, 86].

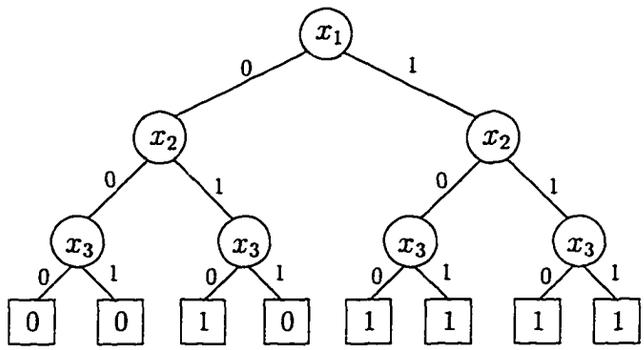
Of particular interest is the relationship between the structure of an ROBDD representing a function and the decompositions that exist in the function. Also of interest are relationships between the autocorrelation of a function and the structure of the function's ROBDD representation. This chapter shows that there are strong relationships between variable ordering, two-place symmetry, two-place decomposition and autocorrelation.

Section 7.1 defines BDDs and ROBDDs. The sections that follow discuss applications of ROBDDs emphasizing spectrum and autocorrelation representation and multi-level synthesis, with particular emphasis on two-place decomposition.

7.1 Binary Decision Diagrams

Binary Decision Diagrams [5, 55] are rooted directed acyclic graphs (DAGs) that are used to represent switching functions. The structure of a BDD is similar to that of a binary tree, but nodes can have more than one incoming edge so that paths are not disjoint. All the leaf nodes are *terminal nodes* and are labeled with either a 0 or a 1, which corresponds to the values that the function can have. Every internal node or *non-terminal node* is labeled by a function variable and has two outgoing edges, labeled 0 and 1, corresponding to the values the variable can take. By convention, the root of the DAG is at the top and all edges are directed downward. For example, Figure 7.1 represents a BDD for the function defined by Table 2.1.

Figure 7.1: BDD Representation of Function in Table 2.1



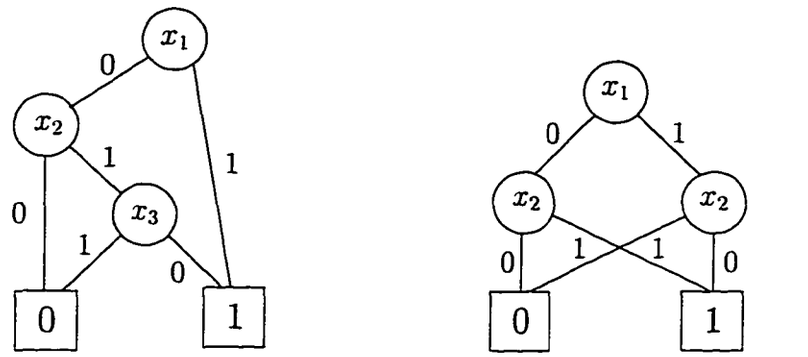
To evaluate the value of a function for a particular input assignment, one follows a path, as determined by the values of the variables, from the root node to a terminal node. From each node, the edge that is followed is the one labeled with the value corresponding to the value assigned to the variable. This leads either to another non-terminal node, in which case another edge is followed, or to a terminal node, the label of which gives the value of the function.

A BDD is termed *ordered* if each variable appears at most once on any path from the root to a terminal node, and the variables adhere to the same fixed order on all paths. A BDD is *reduced* if it contains no redundant nodes, i.e. nodes that have both outgoing edges pointing to the same node, and no pair of nodes labeled with the same variable with equal 0 and equal 1 descendants. A BDD that is ordered and reduced is called a Reduced Ordered Binary Decision Diagram (ROBDD) [17], and is a canonical representation, up to variable order, for a switching function. The ROBDD of Figure 7.2(a) corresponds to the BDD in Figure 7.1.

The ROBDD's popularity is due to several advantages that it has for function representation and manipulation. One advantage is that some algebraic manipulation operations on the functions may be performed more easily with ROBDDs than with other representations [17]. The primary advantage is that many functions can be

Figure 7.2: ROBDD Representations of Functions

(a) ROBDD of Table 2.1 (b) ROBDD of $f = x_1 \oplus x_2$



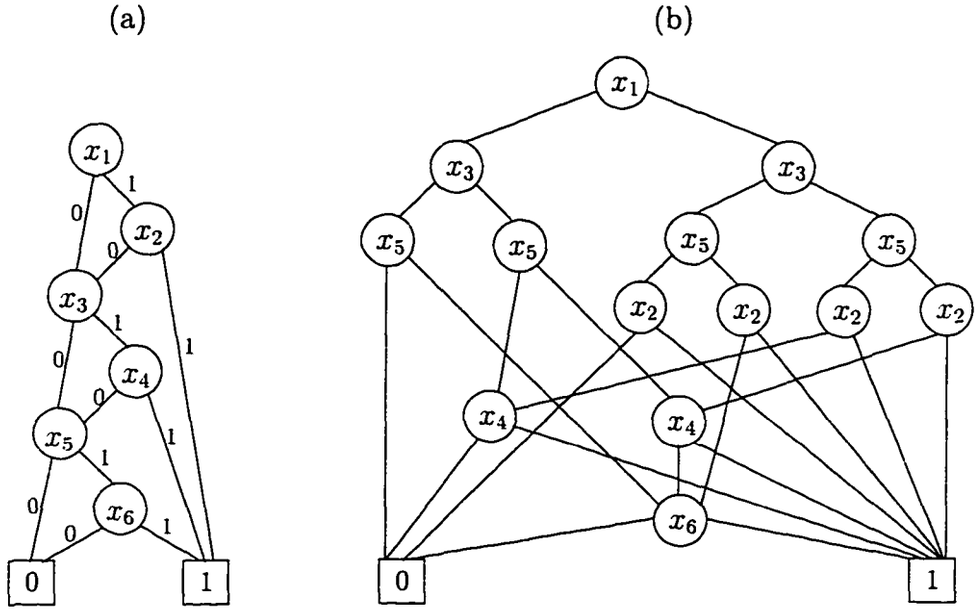
represented much more compactly than by using a truth table or cube list. For example, an Exclusive-OR (XOR) function of n variables requires 2^{n-1} cubes in a cubelist representation (2^n cubes if both the 0's and 1's of the function need to be specified) but requires only $2n + 1$ vertices in a ROBDD representation, as shown in Figure 7.2(b) for $n = 2$.

For a given variable order, the ROBDD representation of a function is unique. However, the number of nodes required in an ROBDD can be highly dependent on the variable order. This is exemplified in Figure 7.3, which is taken from [17]. Both ROBDDs represent the same function, but each has a different variable order.

The number of nodes required in a full BDD representation of a switching function $f(x_1, \ldots, x_n)$ such as the one in Figure 7.1 is $2^{n+1} - 1$, which is double the size required for a truth table. In the worst-case, the number of nodes for an ROBDD is exponential in n. Bryant [17] has shown that at least one of the $2n$ outputs of the n-bit multiplier requires an ROBDD with at least $2^{n/8}$ nodes, regardless of the variable order. However, most practical functions have reasonably sized ROBDDs.

To reduce the size of ROBDDs further, one may use *attributed edges* [66, 68] where

Figure 7.3: Two Variable Orders for $f(x_1, x_2, x_3, x_4, x_5, x_6) = x_1x_2 + x_3x_4 + x_5x_6$
Note: 0 edges on left, 1 edges on right

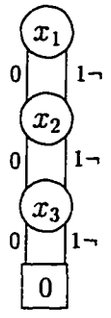


certain operations are attached to the edges of the DAG. Minato et al. [68] discuss various types of edge attribute but the most useful and most often used is the *output inverter* or *output negator*, which is also used in [60].

Every node in a ROBDD is in fact the root of a DAG representing some subfunction. With the use of output negators only a function or its complement need be represented, but not both. A negator is added to each edge in the ROBDD. If it is off it has no effect. If it is on, it means the edge points to the complement of the function that has the node to which the edge points as its root.

Output negators can substantially reduce the size of a ROBDD. For example the ROBDD for the XOR of n inputs has $2n + 1$ nodes without negators but has n+1 nodes with negators. See Figure 7.4 for an example of the XOR of three inputs. Note that in an ROBDD with output negators the two outgoing edges from a node can lead to the same node, but they in fact point to a function and its complement as a

Figure 7.4: Three Variable XOR with Output Negators



result of the negator on the 1-edge. The node is thus not redundant.

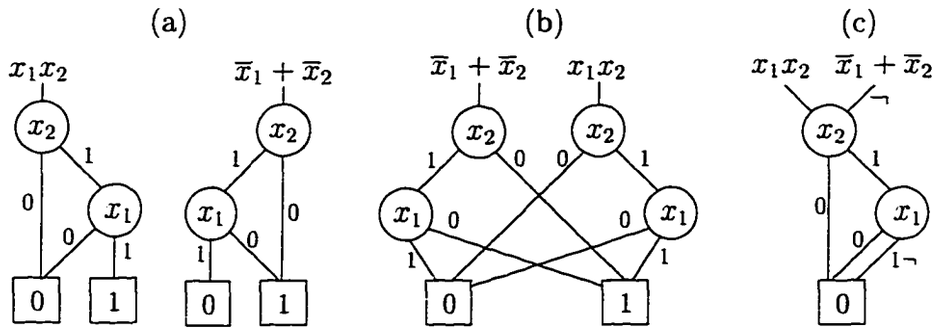
When output negators are used, rules must be enforced to ensure that the uniqueness of the ROBDD representation is preserved. The usual rules are:

- the terminal node (there is only one) has the value 0;
- no 0-edge in the ROBDD has an output negator on it. This ensures that the same choice between representing a function or its complement is always made. An in-going edge must be added to the root since the function may have to be complemented.

Other edge attributes identified by Minato [66, 68] are not so commonly used since more work is required to preserve the uniqueness properties, resulting in overhead for not much gain.

To represent multiple-output functions one may use a *Shared Binary Decision Diagram* (SBDD) which is a multiple-rooted BDD with one root for each function output. This allows for sharing of isomorphic subgraphs among the outputs. Figure 7.5 depicts three ROBDD representations of two two-variable functions. As shown, the number of nodes required to represent the functions decreases from Figure 7.5(a) to Figure 7.5(b) as ROBDDs are shared and then again from Figure 7.5(b) to Figure 7.5(c) as output negators are used.

Figure 7.5: ROBDD Representations of a System of Two Two-variable Functions
(a) separate ROBDDs
(b) shared ROBDDs
(c) shared ROBDDs with output negators



Most of the ROBDD work has been in the context of completely specified functions. Two suggestions for extending the representation to incompletely specified functions are [67, 68]:

1. use a third terminal symbol "-" representing don't-care;
2. represent the incompletely specified function by two completely specified functions, one with the don't-cares set to 0 and the other with the don't-cares set to 1.

The latter has the advantage that since the incompletely specified function is represented by a pair of completely specified functions, no modification to a standard ROBDD package is required. In addition, logic operations are readily implemented.

In particular, a function f is represented by a pair of functions (f_0, f_1) with the encoding

f	f_0	f_1
0	0	0
1	1	1
-	0	1

.

Hence,

$$\begin{aligned}\overline{(f_0, f_1)} &= (\overline{f}_1, \overline{f}_0),\\ (f_0, f_1)(g_0, g_1) &= (f_0 g_0, f_1 g_1),\\ (f_0, f_1) + (g_0, g_1) &= (f_0 + g_0, f_1 + g_1).\end{aligned}$$

Note that the positions of all don't-cares can be readily identified as the true minterms of $\overline{f}_0 f_1$.

Minato [67, 68] has in fact shown that the two methods are equivalent. By the introduction of a new variable D, the f_0 and f_1 required in method (2) can be represented by a single ROBDD as

$$\overline{D} f_0 + D f_1.$$

In this case, D appears at the top of the ROBDD.

If D is reordered to the bottom of the ROBDD, the resulting ROBDD is in fact the representation of method (1) with the terminal symbol "-" replaced by a node labeled D whose 0-descendant is 0 and whose 1-descendant is 1. Minato [67, 68] notes that, as for regular variables, the D variable should appear at the top of the ROBDD if it greatly affects the function. The notion of the D variable is a powerful one in that it unifies the two methods noted above and allows for the notion of don't-cares to be effectively implemented in an ROBDD package designed for totally specified functions.

7.2 Spectral Transforms using Decision Diagrams

Due to the exponential size of function truth tables and spectral transform matrices, computation of a full spectrum or autocorrelation until recently has been limited to functions of less than approximately twenty variables. While relatively efficient techniques using cube lists [30, 91, 93, 94] have been developed for computation

of subsets of spectral or autocorrelation coefficients, they still face the exponential growth problem when attempting to compute the entire spectrum.

Decision diagram techniques thus have been researched to represent, compute and manipulate spectra of switching functions and several have been utilized to develop efficient spectrum computation techniques for large functions [19, 52, 60, 62, 89]. These techniques can be used to compute the full spectrum as well as subsets of the coefficients.

7.2.1 Full Transform Techniques

Miller [60] presented an algorithm for the computation of the spectrum of a switching function directly from an ROBDD representation of the function. The spectrum itself is represented as a reduced ordered decision diagram.

The method in [60] is matrix-based in that the algorithm design is derived from the recursive nature of $\mathbf{T}^n$, the Hadamard transform matrix. The procedure is based on the fast Hadamard transform [39].

Clarke et al. [19] developed another matrix based algorithm for computing spectra from BDDs. They however, treat a switching function as an integer valued function rather than as a binary-valued function. Their method represents both switching functions and the Rademacher-Walsh transform matrix by BDDs and the spectrum is computed by performing, with BDD operations, equivalent matrix operations. Clarke et al. take advantage of the recursive nature of the transform matrices and represent them in size linear in n. With their approach, Clarke et al. have computed the spectra for functions of up to 200 variables.

Lai et al. [52] developed a similar approach, except they went one step further and treat the nodes of the BDD as representing arithmetic functions rather than Boolean functions. They also assign a value attribute to each edge in the BDD and hence call

their BDDs Edge-Valued Binary Decision Diagrams (EVBDDs).

In [52] no representation of a transform matrix is required and no matrix operations are performed to compute the spectrum. The spectral transform is accomplished using addition and subtraction on EVBDDs. The algorithm is however based on the recursive structure of the Hadamard transform and is thus similar to the method due to Miller [60].

To compute the autocorrelation of a switching function, one applies the spectral transform procedure twice as given in Equation (2.11). The first application of the transform procedure computes and represents the spectrum with an ROBDD. Then the terminal nodes, which contain the spectral coefficients, are squared and the transform is performed again. The terminal nodes are then divided by 2^n, which is implemented with a simple bit-shift operation, to yield the autocorrelation coefficients.

7.2.2 Computing Individual Coefficients

Frequently, only a few coefficients are required rather than the entire spectrum or autocorrelation of a switching function. For example, in the PLA pairing technique of Chapter 3, only the first and second-order autocorrelation coefficients are used for variable pair selection. Relatively efficient techniques using cube lists [30, 91, 93, 94] have been developed for computation of subsets of spectral or autocorrelation coefficients. Unfortunately, the cube lists used by these methods still can become unmanageably large.

The methods described in [60] and [19] may be used to compute individual coefficients or small subsets of coefficients using ROBDDs. Clarke et al. [19] use a BDD representation of a transform matrix which may be a complete transform, when computing the entire spectrum, or it may be a partial transform consisting of some

subset of the rows of the full matrix. The transform is carried out using the matrix operations Clarke et al. have defined and the result is a BDD representing the subset of coefficients corresponding to the chosen rows.

More recently Thornton and Nair [89] and Miller [63] developed algorithms to compute individual spectra of switching functions based on the *output probability* of the switching function, which is the probability that the value of the function is 1 given that the input variables are equally likely to be 0 or 1.

The major advantage of the approaches taken in [89] and [63] is that they are not dependent on the recursive structure of the Hadamard transform or any specific transform matrix. Any set of *constituent functions* could be used. This has advantages in the area of synthesis, as discussed later.

As is done in the case of computing the full autocorrelation, the methods in [19] and [60] may be used to compute individual autocorrelation coefficients by applying the transform procedure twice. The methods of [89] and [63] cannot be applied in this manner.

However, subsets of autocorrelation coefficients could be computed directly from an ROBDD in a manner similar to the procedure used in Chapter 3 and in [91]. Since each path from the root to a terminal node in an ROBDD forms a cube that is disjoint from the others, one may obtain a disjoint cube list by traversing all paths. Hence one could enumerate all the cubes and use any procedure that operates on disjoint cubes such as that in [91] and [30, 93, 94] to compute individual autocorrelation or spectral coefficients.

7.3 Relating Variable Ordering, Two-place Symmetry, Two-place Decomposition, and Autocorrelation

The number of nodes required in an ROBDD can be extremely sensitive to the order in which the variables appear. There are $n!$ possible variable orders for an ROBDD representing an n-variable switching function. The selection of a good variable order is likely the most difficult aspect of using ROBDDs in logic design. It is known to be an NP complete problem so the emphasis in the research has been on finding heuristic methods of determining a good order.

A major breakthrough is described in [78], where Rudell introduces the notion of *sifting*, a process by which a variable is moved to a new position in the variable order via the exchange of adjacent variables. As discussed below, the exchange of adjacent variables in an ROBDD is an efficient operation.

Rudell's *sifting algorithm* proceeds as follows. The variables are first sorted into decreasing order according to the number of nodes that each variable labels in the initial ROBDD. For each variable x_i:

1. sift x_i to the bottom of the ROBDD by adjacent variable swapping;
2. sift x_i to the top of the ROBDD by adjacent variable swapping;
3. restore x_i to its best position found in the above sifting by further adjacent variable swaps. The best position is the one that yields the fewest nodes in the ROBDD. In the case of a tie, the one that requires the least reordering is chosen. It can take up to $n-1$ swaps to restore the variable to its best position.

Each variable is moved only once in the process. When variable x_i is being sifted, all other variables remain in a fixed order, with x_i being inserted at the position which

minimizes the size of the ROBDD. Once a variable is placed in position, with all others fixed, it remains in that relative position in the variable order.

Rudell's method chooses the best of n^2 variable orders starting with an initial order that is chosen randomly or by using a heuristic. For each variable it performs an exhaustive search for its best position assuming that the positions of all other variables remain fixed.

There are two important ideas in Rudell's paper with respect to making the method reasonably efficient. First, multiple hash tables should be used, one for each variable, so that all nodes labeled by a particular variable can be accessed without having to traverse the graph. This is an extension to the notion of *threads* that Miller uses in [60].

Second, and very important, is the fact that with some care, two adjacent variables can be exchanged without affecting any nodes higher or lower in the ROBDD. This is crucial since it means that one can scan across the ROBDD exchanging levels i and $i+1$ without having to consider any other levels.

Many have observed that symmetric variables tend to appear together in good variable orders, e.g. [42]. This is the basis of *group sifting* methods [74], which proceed like simple sifting but also determine if two adjacent variables are symmetric. If two symmetric variables are found, they are locked and henceforth sifted as a group. For functions with a high degree of variable symmetry, the improvement over simple sifting is quite substantial.

Panda and Somenzi [73] extend the notion of the group beyond simple variable symmetry. They consider positive symmetry, where a function is symmetric in x_i and x_{i+1}, and negative symmetry, where a function is symmetric in x_i and $\overline{x}_{i+1}$. They also consider mixed symmetry which means that as each node labeled x_i is considered, it must exhibit either positive or negative symmetry with x_{i+1}. Finally they consider almost symmetric pairs by allowing a fixed level of violation of the

symmetry conditions.

A central facet of their method is the check for symmetry. They present the following theorem (which is restated here since the original statement is poorly worded).

Theorem 7.1 *Consider an ROBDD representation of a Boolean function f, in which x_i is immediately before x_j in the variable order. Then f is symmetric in x_i and x_j, if, and only if,*

1) for all nodes labeled x_i, the condition $g_{x_i\bar{x}_j} = g_{\bar{x}_i x_j}$ holds, and

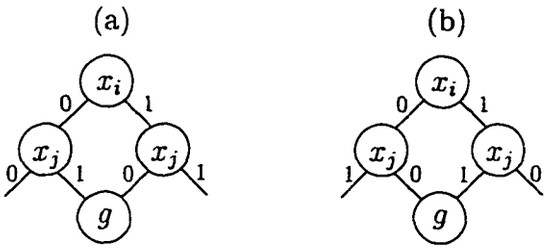
2) all edges leading to nodes labeled x_j come from nodes labeled x_i.

In (1), the two functions are cofactors of the subfunction that has a node labeled x_i as its root. The condition arises from the fact that for the completely specified function to be symmetric in x_i and x_j, all subfunctions found by fixing the variables above x_i in the ROBDD must exhibit the symmetry. As seen below, verifying this condition involves checking certain edge relations in the ROBDD.

Criterion (2) is verified using reference counts. The reference count of a node is the number of incoming edges. In this case, the sum of the reference counts for the nodes labeled x_j must equal the number of edges from nodes labeled x_i to nodes labeled x_j. This is checking for the existence of a subfunction that depends on x_j and not x_i which means x_i and x_j can not be symmetric. The opposite condition, a subfunction which depends on x_i and not x_j, is covered by condition (1).

Figure 7.6(a) shows the check required for each node labeled x_i to verify a positive symmetry in x_i and x_j. As shown, one need only verify that the 0-1 and 1-0 paths from the x_i nodes lead to the same node i.e. the same subfunction. A further check of edge negators is needed if they are used (recall that edge negators never appear on 0-edges). In this case, the 0-1 and 1-0 paths lead to the same function, if, and only

Figure 7.6: ROBDD Structure of Positive and Negative Symmetry in x_i and x_j
(a) Positive symmetry
(b) Negative symmetry



if, the 1-edges from the top and the left node either both do, or do not, have edge negators.

As shown in Figure 7.6(b), the check for negative symmetry is similar (note the labeling of the edges from nodes labeled x_j). In this case the 1-edges from the top and the right node must both have, or not have, edge negators.

The conditions being checked are for two of the six possible two-place symmetries in x_i and x_j. The other four are not considered. However, it appears the other four are in fact stronger in the sense that they effect more compaction in the ROBDD. For example, Figure 7.7 is the required structure for 00-01 symmetry. The 0-0 and 0-1 cofactors must be equal so one node labeled x_j drops out. The other 3 cases are analogous.

The structures in Figure 7.6 are the conditions for the existence of a complex

Figure 7.7: ROBDD Structure for an SND Two-place Decomposition

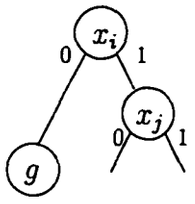
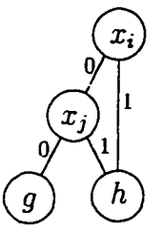


Figure 7.8: ROBDD Structure for an SD Two-place Decomposition



disjunctive decompositions in x_i and x_j (note that the condition has to be satisfied for every node labeled x_i). Figure 7.7 shows the condition for the existence of one of the four possible simple non-disjunctive decompositions.

Combinations of these conditions lead to the requirement for a simple disjunctive decomposition. For example, Figure 7.8 shows the structure that is required for a simple disjunctive decomposition with 00 as the distinguished vertex.

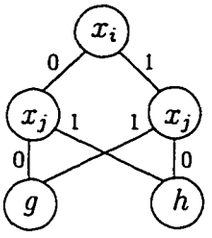
A simple disjunctive decomposition involving XOR has the structure of Figure 7.9 without edge negators.

After studying the various cases it is obvious that if a particular two-place decomposition exists in variables x_i and x_j, then every sub-DAG that has a node labeled x_i as its root must satisfy the following:

decomposition type	x_i nodes	x_j nodes (at most)	subfunctions (at most)
simple disjunctive (non-XOR)	1	1	2
simple disjunctive (XOR)	1	2	2
simple non-disjunctive	1	1	3
complex disjunctive	1	2	3

This is a very interesting hierarchy which corresponds to the preference order used in the two-place decomposition method. It indicates that variables exhibiting a two-place decomposition should likely appear together in a good variable order.

Figure 7.9: ROBDD Structure for an XOR SD two-place decomposition



As mentioned, it had been empirically observed that symmetric variables tend to appear together in good variable orders. Indeed it had been conjectured that there would always be an optimal variable order where symmetric variables appear together.

Möller et al. [70] find this to be the case for most but not all functions of up to 5 variables. Panda and Somenzi [73] provide the following three-variable example:

$$\zeta(x, y, z) = xy + y\overline{z} + \overline{y}z. \tag{7.1}$$

The function is symmetric in y and z but for an ROBDD with edge negators the optimal orders are $y < x < z$ and $z < x < y$ which require 4 nodes whereas the two orders with the symmetric variables y and z adjacent require 5 nodes, the terminal node being included in both cases.

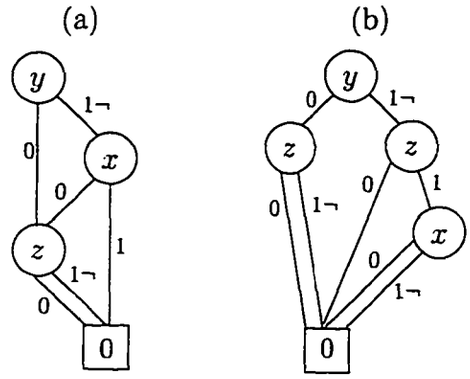
Panda et al. [74] further show that functions exist for which an optimal order has symmetric variables arbitrarily far apart. The example they give is a parameterized function

$$Z_d(x_1, y_1, z_1 ... x_d, y_d, z_d) = \bigoplus_{i=1}^{d} \zeta(x_i, y_i, z_i).$$

This function places the symmetric variables distance d apart with or without edge negations for all $d > 1$. For $d = 1$, it is just the case above for edge negators, but is not valid in the absence of edge negators.

Consider Equation (7.1) again. An optimal order is shown in Figure 7.10(a) while

Figure 7.10: Variable Orders for Panda and Somenzi Example
(a) Optimal variable order
(b) Sub-optimal variable order



a sub-optimal order with y and z adjacent is shown in Figure 7.10(b).

In Figure 7.10(a) there are two descendants from the nodes labeled x and y. Likewise in Figure 7.10(b), there are two descendants from nodes labeled y or z, but in this case there are two nodes labeled z. The difference is that Figure 7.10(a) exhibits a 00-01 symmetry in y and x while Figure 7.10(b) exhibits a 01-10 symmetry in y and z.

There is also a 00-01 symmetry in x and z. Hence the apparent advantage of the orders $y < x < z$ and $z < x < y$ is that there are simple non-disjunctive decompositions in the two adjacent pairs. Orders such as $x < y < z$ and $y < z < x$, where the symmetric variables are adjacent, exhibit one simple non-disjunctive and one complex disjunctive decomposition in adjacent variables.

There is a strong connection between two-place symmetry, two-place decomposition and good variable orders for ROBDDs. In particular, if two variables are adjacent in the ROBDD it is easy to determine the two-place symmetries that exist. Essentially, the more symmetry the simpler the ROBDD with respect to those two variables. Interestingly, the ROBDD structure is the simplest if the symmetries exist

for a simple disjunctive decomposition, more complex for a simple non-disjunctive decomposition, and more complex again for a complex disjunctive decomposition.

The converse should also hold in that a good variable order should maximize the appearance of symmetry in adjacent variables. In particular, not just the positive and negative symmetry considered by Panda et al. [74], but the more general form of symmetry required in the two-place decomposition method.

This means that finding a good variable order is in fact more difficult than finding two-place decompositions, since if a decomposition exists it indicates something about how the variables are related. However, if there is no decomposition, then some measure of "closeness" to a decomposition is required to indicate if the variables should be close together. Hence using autocorrelation to identify a good variable order is much the same as trying to use autocorrelation to predict two-place decompositions.

7.3.1 Autocorrelation and Variable Ordering

The above demonstrates that there is a relation between two-place decomposition and variable ordering in ROBDDs. It also illustrates that using autocorrelation techniques to determine a good ROBDD variable order is likely closely related to using such techniques for predicting the existence of two-place decompositions, in which case the theorems from Chapter 6 might well prove useful.

The determination of a good variable order is a complex problem and studying it in detail is beyond the scope of this dissertation. However, some results due to Crow [21], of using the autocorrelation of a switching function to determine a good variable order, are presented here.

In [21] the first and second-order autocorrelation coefficients of a switching function are used to determine variable orders for switching functions. The orders, as determined by the autocorrelation procedures, are compared to variable orders found

by the multi-level synthesis system Item [46, 47]. The measures used for purposes of comparison are: the number of nodes in the ROBDD representing the function, the number of CLBs in a LUT-based FPGA implementation of the function, and the speed of determining a variable order.

Four different procedures are developed: one based on only the first-order coefficients, two based on the second-order coefficients, and one based on the average of the first and second autocorrelation coefficients.

The results in [21] show that when applied to single-output functions the autocorrelation based variable ordering techniques produce, on average, as good as or better results than Item. The major advantage of the autocorrelation-based methods is that they are considerably faster than Item. This is true even considering that the full autocorrelation is computed even though only the first and second-order autocorrelation coefficients are required. When used for multiple output functions, the autocorrelation methods perform less well, but are still much faster than Item.

The method found to provide the best overall performance is the method based on the average of the first and second-order coefficients for each pair of variables. This value is similar to that used in Chapter 3 for the minimal-variance pairing selection heuristic. This also relates to the theorems of Chapter 6.

7.4 ROBDD Techniques for Non-Two-place Decomposable Functions

As seen in Chapters 5 and 6, when a function exhibits no two-place symmetries, the two-place decomposition procedure alone cannot generate a realization for it. In Chapter 6 one technique of transforming a system of functions with no decompositions into a decomposable system using the total autocorrelation of the system is presented.

Algorithm 7.1: ROBDD-based Decomposition for Decomp

Construct a shared ROBDD using output negators representing the system of completely-specified functions
Use Rudell's sifting method [78] to find a good ordering of the variables.
Repeat until each output is represented by a single variable
{
select one non-terminal node according to the following criteria
(1) select a node with the maximum number of parents
(2) within (1) select a node with maximum support up to a user-defined limit
(3) within (2) select a node which is the root of a sub-DAG of minimal weight
extract the sub-DAG rooted by the selected node
convert the extracted sub-DAG to cube form and apply Decomp to determine a realization for that subfunction
replace the selected node in the ROBDD with a new node labeled by a new variable
}

Here another approach is presented.

Various ROBDD-based function decomposition techniques have been developed for logic synthesis for LUT-based FPGAs [41, 86]. This section presents a new method, based on the method described in [41], of decomposing functions that are not two-place decomposable. The approach is to partition the ROBDD representing the system of functions into subfunctions and then to use Decomp to synthesize them. Algorithm 7.1 details the procedure.

Algorithm 7.1 partitions a given system of completely specified functions into a number of single-output completely specified subfunctions such that determining realizations for each of these subfunctions yields a realization for the original system.

To do this, the algorithm identifies a single node in the ROBDD according to three heuristic selection rules. That node serves as the root of a sub-DAG representing a particular subfunction.

A cube specification is determined from the sub-DAG which has the selected node as its root, and Decomp is used to find a circuit realization. The selected node, and the corresponding sub-DAG, are replaced by a single node labeled by a new variable representing the output of the circuit realization. Note that standard reference count techniques are used so that when the sub-DAG is removed, any nodes required elsewhere in the ROBDD are retained.

This process iterates, selecting a sub-DAG and finding a circuit realization for it, until each function in the original system is represented by a single variable, which is the output of the circuit that realizes that function. The key to the process is of course the selection of appropriate nodes.

Three heuristics, based on the rules given in [41], are used to govern the selection. In [41], the heuristics stress the minimization of the number of subfunctions since LUT-based FPGA synthesis is the subject of the work. However, those heuristics are here modified to emphasize sharing of gates.

The rationale for the heuristics is as follows:

1. A node with as many parents as possible is selected. Since the selected node is replaced by a new node representing the circuit synthesized for the selected subfunction, all nodes pointing to the selected node share the realized circuit. Hence a node with as many parents as possible is selected to improve the sharing of gates.

2. Within (1) select a node with maximum support up to a user-defined limit. The support of a node is the number of inputs to the subfunction for which the node is the root. Due to the structure of a BDD node, the limit should be set at three

or greater since, excepting the case of a node with an edge leading to a terminal, each BDD node represents a function involving the variable representing the node and two distinct descendants. The larger the support, the larger the extracted subfunction, which can be of advantage. However, it also increases the possibility that Decomp can not deal with the selected subfunction.

Experience shows that limiting the extracted subfunctions to five inputs or less works well. In particular, when the limit is raised to six, subfunctions are extracted that can not be decomposed even with linearization. Interestingly, this corresponds to experimental evidence that five is an optimal size for LUTs in FPGAs [16]. Also, it is known that all but two of the spectral classes of functions of up to five variables are two-place decomposable [40] and all spectral classes of functions of up to four variables are two-place decomposable. Therefore, even if a selected subfunction is not two-place decomposable, it can be made decomposable using linearization.

3. Within (2), select a node which is the root of a sub-DAG of minimal weight. Assuming a reasonably good variable ordering, the number of nodes in an ROBDD (its weight) is a good estimate of the complexity of the function it represents. Thus this heuristic attempts, within the other criteria, to select a simple subfunction.

The above procedure successfully synthesizes several benchmark functions. Following are the details of applying the method to *mlp4*, which takes as input two 4-bit integer values and produces their 8-bit product. The multiplier is a good example to use as it is well known to cause problems for ROBDD based techniques [17] and it is also problematic for Decomp.

For this example the variables are labeled a_3, a_2, a_1, and a_0 corresponding to the bits of the first integer from most to least-significant. Similarly b_3, b_2, b_1, and b_0 are the variables for the second integer.

The number of nodes required in the initial ROBDD using the variable order, $a_3 < a_2 < a_1 < a_0 < b_3 < b_2 < b_1 < b_0$, is 140. After sifting, the final ROBDD using order $a_3 < a_2 < a_1 < a_0 < b_0 < b_3 < b_2 < b_1$, has 135 nodes. Note that the two least-significant bits are brought together as the functions are symmetric in those two inputs.

The ROBDD decomposition identifies 80 functions to be decomposed as follows:

inputs	number of functions
2	13
3	19
4	12
5	36
total	80

Note the 2-input functions are trivial but it is simpler just to pass them along to Decomp and not treat them as a special case.

The functions are decomposed using the command (refer to Appendix A for the syntax of the Decomp command):

```
decomp -lf -r
```

Note the use of the linearization option for functions that are not two-place decomposable.

The decomposition produces circuits with a total of 382 gates. One five gate prefilter is generated. However, following merging of the circuits and removal of redundant gates, the circuit has 294 gates and 18 levels.

Table 7.1: ROBDD + Decomp Synthesis Results

	#	ROBDD Decomp		MIS	
func	pi/po	gates	levels	gates	levels
postal	8/1	34	10	26	9
mlp3	6/6	63	9	53	9
mlp4	8/8	294	18	219	25
sqr5	5/10	69	10	43	7
sqr6	6/12	133	11	98	24
alu2	10/8	168	18	60	8
bw	5/28	203	11	148	7
clip	9/5	176	15	109	14
con1	7/2	16	6	17	5
duke2	22/29	517	18	540	14
misex1	8/7	64	8	39	5
misex2	25/18	92	7	82	8
misex3	14/14	1009	25	752	15
sao2	18/4	162	14	126	11
vg2	25/8	322	21	62	11

The CPU time required on a SUN SparcServer 20 is:

operation	time (sec)
build ROBDD, sift and extract functions	0.5
decompose all functions	6.7
merge circuits and remove duplicates	0.1
total	7.3

Table 7.1 provides results for several benchmark functions. For comparison purposes, data regarding MIS generated circuits are included in the table. For a fairer comparison, the statistics for the MIS column are obtained by using the circuit reduction algorithm from Decomp on the two-input-gate circuit generated by MIS.

As seen in Table 7.1 the combination of Decomp with Algorithm 7.1 generally does not perform as well as MIS. All but two of the benchmark circuits are larger, one by a huge amount. e.g. 519% larger for *vg2*. However, the rest of the results are

in the range of only 12% to 64% larger. Also, although most realizations have more gates than MIS's circuits, several of those have many fewer levels. For example, the MIS circuits for *mlp4* and *sqr6* have 39% and 46%, respectively more levels than the circuits synthesized by Algorithm 7.1.

7.5 Conclusion

This chapter considers some applications of ROBDDs in combinational logic synthesis. It shows that there are strong relationships between variable ordering, two-place symmetry, two-place decomposition and autocorrelation. As well, a new synthesis procedure, merging an ROBDD decomposition technique with two-place decomposition, is presented. This procedure can be used as an approach for synthesizing functions that are not two-place decomposable.

The results given in Table 7.1 show that this is a promising technique since it produces some good results and allows Decomp to be applied to a broader range of functions. The true power of this approach comes from Decomp. The ROBDD decomposition is based on [41] and is quite simple. Sifting is a well known and widely used technique. It is combining these with Decomp that works well.

Note that this technique presently can be used only for completely specified functions. Therefore it can be used during the two-place decomposition procedure to convert a non-decomposable image function into a decomposable image function only if it is completely specified.

The results may be improved by using a more sophisticated variable ordering method than sifting. For example one could use group sifting where the symmetries that are checked include all six possible symmetries. The variables within the group would be ordered according to the two-place decomposition preference ordering.

The result of Algorithm 7.1 is a set of decomposed functions. At present the func-

tions are decomposed separately. This could likely be improved by forming multiple-output problems, but note that node selection rule (1) of the algorithm takes care of some of the sharing and a simple merger and duplicate gate removal process is quite effective.

Möller et al. [69] use a traversal of the ROBDD and some clever checks to detect pairs of positive and negative symmetric variables i.e. 01-10 and 00-11 symmetries. This method is readily extended to include checks for the other four symmetries. But the approach is inherently limited to totally specified functions because it simply checks for edges leading to the same function to confirm function equivalence. Such an approach does not extend to the case of function compatibility as required for checking symmetry in partially specified functions.

Panda and Somenzi [73] check for symmetry during sifting. Here too only 01-10 and 00-11 symmetry is checked, but the method can be extended to include checks for all six symmetries. Again, the method is applicable only to completely specified functions.

A topic for further consideration is the application of the cube-based symmetry checks that are performed in Decomp. Since the paths in an ROBDD correspond to disjoint cubes, the cube-based symmetry checks that are done can be accomplished by considering paths in the ROBDD and, unlike the above techniques, is extendible to functions with don't-cares. A difficulty would be in arranging to look only at necessary pairs of paths (cubes).

The results presented here may improve also if the ROBDD decomposition and two-place decomposition procedures interacted. At present the ROBDD decomposition and two-place decomposition are treated as two disjoint steps. One could imagine a more sophisticated approach where two-place decomposability is used in selecting the decomposition nodes in the ROBDD. However, as shown in Chapters 5 and 6 it is difficult to predict when functions are decomposable.

Chapter 8

Conclusion

This dissertation considers a variety of approaches to the synthesis and optimization of combinational logic. Methods for both structured (PLA-based) and random (gate-based) circuits are examined.

8.1 Major Results

The following describes the problem areas that are addressed in this dissertation and summarizes the major results.

8.1.1 PLA Optimization

Chapter 3 introduces a new approach for finding near-optimal variable pairings for decoded PLAs. The pairing procedure uses information contained in the total autocorrelation of a system of switching functions, particularly the information contained in the first and second-order autocorrelation coefficients. Two variants of the basic algorithm are developed to handle functions for which the basic algorithm does not perform well.

Using certain benchmark functions, the new approach is compared with techniques used by Sasao [81, 82, 84], and by Chen and Muroga [18]. As seen in Chapter 3, the autocorrelation pairing procedure yields results that are comparable to that of these other approaches.

The autocorrelation pairing procedure selects optimal pairings for 20 of the 40 functions for which exhaustive search data are listed. For all but four of the remaining functions, the pairing selected is within five product terms of the optimum.

The autocorrelation pairing results compare favourably with those of Sasao. Of the 41 benchmark functions used, the autocorrelation procedure produces better results in 7 cases and worse results in 10 cases with a variation of more than two product terms for only five functions. As mentioned, a variation of one or two product terms is not necessarily significant due to heuristics used in the PLA minimization procedures.

For arithmetic functions, the autocorrelation procedure is found to perform well since these function exhibit significant XOR structure. For control functions, which exhibit much less XOR structure, the autocorrelation pairing procedure does not perform as well. However, some of these results are improved by using a variant of the basic approach.

The comparison with Chen and Muroga's algorithm is similar. The autocorrelation procedure produces results that are better than those of Chen and Muroga for two of the eleven reported results and worse results for three. Except for one function, the variation between the results is at most three product terms.

For most of a set of large functions, Sasao's algorithm selects better pairings. However some of the results are improved by using variants of the basic autocorrelation algorithm. Of more significance are the timings. The timings show that the autocorrelation approach can be considerably more efficient than Sasao's approach.

As described in Chapter 3, the key advantages of the new approach are its relative simplicity and its efficiency. The procedure's speed allows the user, or another logic

optimization or synthesis tool, to perform more exploration for an acceptable solution by trying any or all of the variants of the autocorrelation pairing procedure and use the best results that are obtained.

Procedures based on the pair selection autocorrelation procedure are developed to select larger groupings. In particular triple and 1-triple pair assignments are selected. However, the procedures do not perform as well for these larger groupings.

PLA optimization can also be applied to FPGA logic synthesis for switching circuits to be realized with PLD-based FPGAs such as those produced by Altera [6] and AMD [4]. This optimization may be used to fit a system of switching functions more efficiently into such an FPGA. It is also suggested in Chapter 3 that a new type of FPGA, one with both decoder cells and PLD-based logic blocks, could be developed to take maximum advantage of this type of optimization.

8.1.2 Two-Place Decomposition

Chapter 5 presents two-place decomposition, describes several enhancements to the basic procedure, and introduces Decomp, a robust implementation of the basic two-place decomposition method together with the enhancements introduced in this dissertation.

Enhancements described in Chapter 5 include a better approach to decomposition merging for multiple-output functions, utilization of an XOR mapping function in complex disjunctive decompositions, which is observed to be a case of linear translation of the original complex disjunctive mapping functions, and post processing circuit reduction to combine the two-input gates into larger circuit elements (assuming CMOS type gates). This last enhancement performs a simple technology mapping step and produces fairly good results.

In Chapter 5, Decomp is compared with other multi-level synthesis systems. For

symmetric functions, Decomp is found to perform considerably better than Oasis [15], MIS [14], and a symmetric synthesis procedure developed by Kim and Dietmeyer [51]. For arithmetic functions that exhibit significant symmetry, Decomp again outperforms Oasis and MIS. However, for arithmetic functions that exhibit little symmetry and for control functions, which also exhibit little symmetry, Decomp generates larger realizations and in several cases is not able to complete due to the generation of non-decomposable image functions.

Chapter 6 extends the two-place decomposition procedure by incorporating linearization using the total autocorrelation of a system of functions.

The effects of linearizing functions prior to any synthesis is analyzed and it is found that although the linearization process tends to reduce the size of two-level realizations of functions, it has little benefit for multi-level realizations. Indeed, in the vast majority of cases, it increase the sizes of multi-level realizations generated by Decomp. Thus linearization should be used only when an image function is not two-place decomposable.

When linearization is used during the decomposition process to attempt to convert non-decomposable image functions into decomposable ones, Decomp is able to realize many more functions. However, these realizations are not as good as those generated by MIS [14].

The chapter establishes relationships between values of certain of the autocorrelation coefficients when particular two-place decompositions exist in a function. This is the first attempt to formally analyze the relation between autocorrelation and two-place decomposition; hence, the relationships are novel and may lead to a linearization-like procedure oriented to two-place decomposition. They may also prove valuable for decomposition selection and mapping selection.

An approach to handling don't cares in autocorrelation computation targeted to two-place decomposition is proposed in Chapter 6. This approach derives two com-

pletely specified functions from the original, and the autocorrelation of the original function is computed as the cross-correlation of those two completely specified functions.

The circuits generated by Decomp are composed of two-input gates and inverters. Thus they are well suited for implementation with fine-grained FPGAs such as Motorola's MPA1000 FPGA family [71] and Xilinx's XC8100 FPGA family [99]. Alternatively, a technology mapping step may be performed to map the Decomp realization onto other structures. Decomp performs such an process in its circuit reduction operation.

8.1.3 ROBDD Techniques

Chapter 7 presents a new ROBDD-based synthesis technique targeted to two-place decomposition. A procedure based on that developed in [41] for LUT-based FPGA synthesis is utilized by this new technique to partition an ROBDD into subfunctions, which are synthesized by Decomp. The ROBDD-based procedure enables the synthesis of functions for which a realization cannot be generated by two-place decomposition alone, and it allows Decomp to handle large problems since a large problem is partitioned into several smaller ones.

The results of this new synthesis algorithm are compared with results generated by MIS [14] and it is found that for all but two benchmark functions synthesized by the two methods, the realizations generated by the new procedure are in the range of 12% to 64% larger than the realizations generated by MIS. However, several of the realizations have many fewer levels than MIS's realizations.

Although the results are not as good as MIS's in most cases, the method has considerable promise and enhancements to the basic procedure are suggested for improvement.

Chapter 7 also establishes relationships between two-place symmetries, two-place decompositions, optimal ROBDD variable ordering, and the autocorrelation. The results show that the variables involved in a two-place decomposition should be placed together in an ROBDD. The order of preference for decomposition types is the same as that used in the two-place decomposition procedure, i.e. simple disjunctive is chosen first, then simple non-disjunctive, and finally complex disjunctive.

8.2 Future Work

Future work in PLA optimization includes continuing the work into successfully applying the autocorrelation to finding more general optimal groupings for decoded PLAs.

A good measure of "closeness" to a decomposition is desirable. The theorems of Chapter 6 provide some initial insight into how the autocorrelation may be used to determine a closeness measure. This measure can be used in an ROBDD variable ordering algorithm when no two-place symmetries exist in a function. It would also be of great benefit in the PLA pairing optimization of Chapter 3. In fact, the computation used in the minimal-variance heuristic is an attempt to determine such a closeness measure. The rationale for doing so is that the closer a function is to having a simple disjunctive decomposition in variables x_i and x_j, the more likely those variables should be paired since more of the function can be specified with a single product term.

A two-place decomposability closeness measure can also be used in a Thornton-and-Nair-like [89] synthesis approach when no decompositions exist in a function. Here the "error function" would be the difference between the function to be synthesized and the function expressing a particular decomposition.

8.3 Final Remarks

Much of the work presented in this dissertation has practical value as it is applicable to current technologies such as FPGAs. In particular, the work of Chapter 3 is applicable to PLD-based FPGAs and the work in Chapters 5, 6, and 7 is applicable to fine-grained sea-of-gates FPGAs. Given that Motorola and Xilinx, one of the largest manufacturers of FPGA products, both produce fine-grained FPGAs, two-place decomposition may well prove to be a viable approach to practical logic synthesis since the realizations generated by the two-place decomposition procedure can be mapped directly onto the XC8100 [99] and the MPA1000 [71] FPGA families.

The results in this work indicate that there are strong relationships between two-place symmetry, two-place decomposition and optimal variable ordering in ROBDDs. When a system of functions exhibits no two-place decompositions, a measure of closeness to a decomposition is identified as a valuable measure to guide algorithms for the solution of problems ranging from variable pair selection in PLA optimization, to function splitting, to ROBDD variable ordering. While at present it seems such a measure is difficult to define, further work is needed because of the impact a positive result would have across many aspects of logic design.

Bibliography

[1] S. Aborhey, "Autocorrelation testing of combinational circuits," *IEE Proceedings*, vol. 136, Pt. E, Jan. 1989, pp. 57–61.

[2] *ACT 1 Family Gate Arrays, Design Reference Manual.*

[3] *Actel FPGA Data Book and Design Guide.* Actel Corp., 1994.

[4] *MACH 1 and MACH 2 Device Families Preliminary Data Sheets.* Advanced Micro Devices, 1990.

[5] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. C-27, June 1978, pp. 509–516.

[6] *Altera Data Book.* Oct. 1990.

[7] R. L. Ashenhurst, "The decomposition of switching functions," *Annals of the Harvard Computation Laboratory*, vol. 29, 1959, pp. 74–116.

[8] K. Bartlett, R. K. Brayton, G. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Multilevel logic minimization using implicit don't cares," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. CAD-7, June 1988, pp. 723–740.

[9] K. Bartlett, W. Cohen, A. J. De Geus, and G. Hachtel, "Synthesis and optimization of multilevel logic under timing constraints," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. CAD-5, Oct. 1986, pp. 582–595.

[10] K. G. Beauchamp, *Walsh Functions and their Application.* London: Academic Press, 1975.

[11] K. G. Beauchamp, *Applications of Walsh and Related Functions with an Introduction to Sequency Theory.* London: Academic Press, 1984.

[12] K. G. Beauchamp, *Transforms for Engineers (A Guide to Signal Processing).* Oxford: Clarendon Press, 1987.

[13] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis.* Boston: Kluwer Academic Publishers, 1984.

[14] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. CAD-6, Nov. 1987, pp. 1062–1081.

[15] F. Brglez, K. Kozminski, D. Reeves, and G. Kedem, "Simple design examples with OASIS - an Open Architecture Silicon Implementation Software," Tech. Rep., MCNC, 1990.

[16] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays.* Boston: Kluwer Academic Publishers, 1992.

[17] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, Aug. 1986, pp. 677–691.

[18] K.-C. Chen and S. Muroga, "Input assignment algorithm for decoded-PLA's with multi-input decoders," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, Nov. 1988, pp. 474–477.

[19] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large Boolean functions with applications to technology mapping," in *Proceedings of the ACM/IEEE Design Automation Conference*, 1993, pp. 54–60.

[20] R. P. Coleman, "Orthogonal functions for the logical design of switching circuits," *IEEE Transactions on Computers*, vol. EC-10, 1961, pp. 379–383.

[21] J. Crow, *Variable Ordering for ROBDD-based FPGA Logic Synthesis.* Master's thesis, University of Victoria, 1995.

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

[23] M. R. Dagenais, "McBOOLE: A new procedure for exact logic minimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. CAD-5, Jan. 1986, pp. 229–238.

[24] S. Devadas, A. R. Wang, A. R. Newton, and A. Sangiovannin-Vincentelli, "Boolean decomposition in multi-level logic optimization," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, Nov. 1988, pp. 290–293.

[25] L. D. Diaz-Olavarrieta, *Mapping-Based Synthesis using the Goal Oriented Method.* PhD thesis, University of Toronto, Sept. 1988.

[26] L. D. Diaz-Olavarrieta and S. G. Zaky, "Goal-oriented synthesis of switching functions," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, July 1988, pp. 1855–1859.

[27] D. L. Dietmeyer, *Logic Design of Digital Systems.* Bostan, MA: Allyn and Bacon, 2nd ed., 1978.

[28] C. R. Edwards, "The application of the Rademacher-Walsh transform to Boolean function classification and threshold logic synthesis," *IEEE Transactions on Computers*, vol. C-24, Jan. 1975, pp. 48–62.

[29] C. R. Edwards and S. L. Hurst, "A digital synthesis procedure under function symmetries and mapping methods," *IEEE Transactions on Computers*, vol. C-27, no. 11, 1978, pp. 985–997.

[30] B. J. Falkowski and M. A. Perkowski, "Calculation of the Rademacher-Walsh spectrum from a reduced representation of Boolean functions," in *Proceedings of the European Design Automation Conference*, Sept. 1992, pp. 181–186.

[31] R. J. Francis, J. Rose, and Z. Vranesic, "Chortle-crf: Fast technology mapping for lookup-table based fpgas," in *Proceedings of the ACM/IEEE Design Automation Conference*, June 1991, pp. 227–233.

[32] D. Gregory, K. Bartlett, A. J. De Geus, and G. Hachtel, "Socrates: a system for automatically synthesizing and optimizing combinational logic," in *Proceedings of the 23rd ACM/IEEE Design Automation Conf.*, 1986, pp. 79–85.

[33] H. F. Harmuth, *Transmission of Information by Orthogonal Functions.* New York: Springer-Verlag, 1972.

[34] M. Harrison, *Introduction to Switching and Automata Theory.* New York: McGraw Hill, 1965.

[35] S. L. Hight, "Complex disjunctive decomposition of incompletely specified Boolean functions," *IEEE Trans. Elec. Comp.*, vol. EC-22, 1973, pp. 103–110.

[36] S. J. Hong, R. G. Cain, and D. L. Ostapko, "MINI: A heuristic approach for logic minimization," *IBM Journal of Research and Development*, Sept. 1974, pp. 443–458.

[37] S. L. Hurst, *The Logical Processing of Digital Signals.* New York: Crane-Russak, 1978.

[38] S. L. Hurst, D. M. Miller, and J. C. Muzio, "Spectral method of Boolean function complexity," *Electronics Letters*, vol. 18, June 1982, pp. 572–573.

[39] S. L. Hurst, D. M. Miller, and J. C. Muzio, *Spectral Techniques in Digital Logic.* London and Orlando: Academic Press Inc. Ltd., 1985.

[40] B. Iyer, *Classification and Digital Synthesis Using Spectral Techniques.* Master's thesis, University of Victoria, 1984.

[41] R. P. Jacobi and A.-M. Trullemans, "ROBDD-based logic decomposition techniques," in *Proceedings IFIP Workshop on Logic and Architecture Synthesis.* Dec. 1994, pp. 17–25.

[42] S. W. Jeong, T.-S. Kim, and F. Somenzi, "An efficient method for optimal BDD ordering computation," in *International Conference on VLSI and CAD (ICVC'93)*, Nov. 1993.

[43] M. Karnaugh, "The map method for the synthesis of combinational logic circuits," *Transactions of the AIEEE*, vol. 72, 1953, pp. 593–598.

[44] R. M. Karp, "Functional decomposition and switching circuit design," *SIAM Journal on Computing*, vol. 11, June 1963, pp. 291–335.

[45] R. M. Karp, F. E. McFarlin, J. P. Roth, and J. R. Wilts, "A computer program for the synthesis of combinational switching circuits," in *Proceedings of the AIEE Symposium on Switching Circuit Theory and Logic Design*, 1961, pp. 152–162.

[46] K. Karplus, "ITEM: an If-Then-Else Minimizer for logic synthesis," in *Proceedings of Euroasic 1989*, 1989.

[47] K. Karplus, "Xmap: a technology mapper for table-lookup field-programmable gate arrays," in *Proceedings of the ACM/IEEE Design Automation Conference*, June 1991, pp. 240–243.

[48] M. G. Karpovsky, *Finite Orthogonal Series in the Design of Digital Devices.* New York: John Wiley and Son, 1976.

[49] M. G. Karpovsky and E. S. Moskalev, "Realization of partially-defined Boolean functions by expansions in orthogonal series," *Automation and Remote Control*, vol. 31, Aug. 1970, pp. 1278–1288.

[50] B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* New Jersey: Prentice-Hall, 1978.

[51] B.-G. Kim and D. L. Dietmeyer, "Multilevel logic synthesis of symmetric switching functions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. CAD-10, Apr. 1991, pp. 142–157.

[52] Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula, "EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. 13, Aug. 1994, pp. 959–975.

[53] R. J. Lechner, "A transform approach to logic design," *IEEE Transactions on Computers*, vol. C-19, July 1970, pp. 627–640.

[54] R. J. Lechner, "Harmonic analysis of switching functions," in *Recent Developments in Switching Theory*, (A. Mukhopadhay, ed.), New York: Academic Press, 1971.

[55] C. Y. Lee, "Representation of switching circuits by binary decision diagrams," *Bell Systems Technical Journal*, vol. 38, July 1959, pp. 985–999.

[56] R. Lisanke, F. Brglez, and G. Kedem, "McMap: A fast technology mapping procedure for multi-level logic synthesis," *IEEE Transactions on Computers*, 1988, pp. 252–256.

[57] P. A. Lynn, *An Introduction to the Analysis and Processing of Signals*. London: Macmillan Education, 1989.

[58] D. M. Miller, *Decomposition in Many-Valued Logic Design*. PhD thesis, University of Manitoba, Mar. 1976.

[59] D. M. Miller, "A decomposition technique for the design of internally-unate combinational networks," *Congressus Numerantium*, vol. 51, Mar. 1986, pp. 217–232.

[60] D. M. Miller, "Graph algorithms for the manipulation of Boolean functions and their spectra," *Congressus Numerantium*, vol. 57, Mar. 1987, pp. 177–199.

[61] D. M. Miller, "Computing autocorrelation coefficients from cubes," Tech. Rep., University of Victoria, 1991.

[62] D. M. Miller, "Spectral transformation of multiple-valued decision diagrams," in *Proceedings of the International Symposium on Multiple-Valued Logic*, May 1994, pp. 89–96.

[63] D. M. Miller, "An improved method for computing a generalized spectral coefficient," Dec. 1995. Submitted to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[64] D. M. Miller and J. C. Muzio, "Detection of symmetries in totally specified or partially specified combinational functions," *Computers and Digital Techniques*, vol. 2, Oct. 1979, pp. 203–209.

[65] D. M. Miller and J. C. Muzio, "Spectral techniques for fault detection in combinational logic," in *Spectral Techniques and Fault Detection*, (M. G. Karpovsky, ed.), Orlando, Florida: Academic Press, 1985.

[66] S. Minato, "Graph-based representations of discrete functions," in *Proceedings of IFIP WG 10.5 Workshop on the Application of Reed-Muller Expansion in Circuit Design*, 1995, pp. 1–10.

[67] S. Minato, *Binary Decision Diagrams and Applications for VLSI CAD.* Kluwer Academic Publishers, 1996.

[68] S. Minato, N. Ishiura, and S. Yajima, "Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation," in *Proceedings of the ACM/IEEE Design Automation Conference*, 1990, pp. 52–57.

[69] D. Möller, J. Mohnke, and M. Weber, "Detection of symmetry of Boolean functions represented by ROBDDs," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1993, pp. 680–684.

[70] D. Möller, P. Molitor, and R. Drechsler, "Symmetry based variable ordering for ROBDDs," in *IFIP Workshop on Logic and Architectural Synthesis*, Dec. 1994, pp. 47–53.

[71] *The Motorola MPA1000 Fine-Grain FPGA Family.* Motorola Inc., 1994.

[72] J. C. Muzio and S. L. Hurst, "The computation of complete and reduced sets of orthogonal spectral coefficients for logic design and pattern recognition purposes," *Computer and Electronics Engineering*, no. 5, 1978, pp. 231–249.

[73] S. Panda and F. Somenzi, "Who are the variables in your neighborhood?," in *IEEE/ACM Workshop on Logic Synthesis*, May 1995, pp. 5.11–5.20.

[74] S. Panda, F. Somenzi, and B. F. Plessier, "Symmetry detection and dynamic variable ordering of decision diagrams," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, Nov. 1994, pp. 628–631.

[75] J. Poswig, "Disjoint decomposition of Boolean functions," *IEE Proceedings*, vol. 138, Jan. 1991, pp. 48–56.

[76] H. Rademacher, "Einige Sätze über Reihen von Allgemeinen orthogonal funktionen," *Math. Annen.*, no. 87, 1922, pp. 112–138.

[77] J. P. Roth and R. M. Karp, "Minimization over Boolean graphs," *IBM Journal of Research and Development*, vol. 6, 1962, pp. 227–238.

[78] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, Nov. 1993, pp. 42–47.

[79] R. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," in *Proceedings of the International Symposium on Multiple-Valued Logic*, May 1987, pp. 198–207.

[80] A. Saldanha and R. H. Katz, "PLA optimization using output encoding," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, Nov. 1988, pp. 478–481.

[81] T. Sasao, "Multiple-valued decomposition of generalized Boolean functions and the complexity of programmable logic arrays," *IEEE Transactions on Computers*, vol. C-30, Sept. 1981, pp. 635–643.

[82] T. Sasao, "Input variable assignment and output phase optimization of PLA's," *IEEE Transactions on Computers*, vol. C-33, Oct. 1984, pp. 879–894.

[83] T. Sasao, "Functional decomposition of PLA's," in *International Workshop on Logic Synthesis*, May 1987.

[84] T. Sasao, "Multiple-valued logic and optimization of programmable logic arrays," *IEEE Computer*, vol. 21, Apr. 1988, pp. 71–80.

[85] T. Sasao, Personal Communication, Sept. 1991.

[86] T. Sasao, "FPGA design by generalized functional decomposition," in *Logic Synthesis and Optimization*, (T. Sasao, ed.), Boston: Kluwer Academic Publishers, 1993.

[87] S. C. Tai, M. W. Du, and R. C. T. Lee, "A transformational approach to synthesizing combinational circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. CAD-10, Mar. 1991, pp. 286–295.

[88] *Texas Instruments FPGA Data Manual.* Texas Instruments Inc., 1993.

[89] M. A. Thornton and V. S. S. Nair, "Efficient calculation of spectral coefficients and their application," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. 14, Nov. 1995, pp. 1328–1341.

[90] R. Tomczuk, "Combinational logic synthesis by two-place decomposition and autocorrelation techniques," in *Proceedings of the Canadian Conference on Very Large Scale Integration*, Oct. 1992, pp. 139–146.

[91] R. Tomczuk and D. M. Miller, "Autocorrelation techniques for multi-bit decoder PLA's," in *Proceedings of the International Symposium on Multiple-Valued Logic*, May 1992, pp. 355–364.

[92] D. Varma and E. A. Trachtenberg, "A design automation tool for fast, efficient decomposition of logic functions," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, (Santa Clara), 1987, pp. 70–73.

[93] D. Varma and E. A. Trachtenberg, "Design automation tools for efficient implementation of logic functions by decomposition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. 8, Aug. 1989, pp. 901–916.

[94] D. Varma and E. A. Trachtenberg, "Efficient spectral techniques for logic synthesis," in *Logic Synthesis and Optimization*, (T. Sasao, ed.), Boston: Kluwer Academic Publishers, 1993.

[95] J. S. Wallis, "Hadamard matrices," in *Lecture Notes No. 292*, New York: Springer-Verlag, 1972.

[96] J. L. Walsh, "A closed set of orthogonal functions," *American Journal of Mathematics*, vol. 45, 1923, pp. 5–24.

[97] P. Walsh, "Recomposition procedure incorporating inverter minimization.", Personal Communication.

[98] *The Programmable Gate Array Data Book.* Xilinx Inc., 2100 Logic Drive, San Jose, California, 95124, 1992.

[99] *Xilinx XC8100 FPGA Family Data Sheet.* Xilinx Inc., 2100 Logic Drive, San Jose, California, 95124, Oct. 1995.

[100] *Proceedings of the IEEE International Conference on Computer-Aided Design*, Nov. 1988.

Appendix A

Description of the Decomp Command

The Decomp command format is as follows:

decomp [-q{oxq}] [-l[f1]] [-dn] [rusxnvh] [-o output-file] [input-file]

The following are the options that are available:

q - specify the mapping for a complex disjunctive decomposition. Sub-arguments are:

- v - always use AND/OR
- o - always use OR/XOR
- x - always use AND/XOR
- q - try to choose one intelligently (default)

l - linearize the circuit if no decomposition is found. Sub-arguments are:

- f - do full linearization
- 1 - move only one (the largest) coefficient

r - do network reduction post-processing

d - set debugging on and set debug level to "n" (int value)

t - ignore any translation information in the input file

u - generate the dual

s - use single decompositions - do not combine

x - do not allow XOR decompositions

n - apply only one decomposition from those found for an image

o - specify an output file name

v - display version information

h - display a brief help message describing the command syntax options

Appendix B

Example Decomp Traces

This appendix provides example traces of Decomp's execution for a single output function and for three multiple-output functions.

B.1 2-out-of-5 Checker

Function *2of5* is five-input single-output function that has the value 1 if, and only if, two of its five inputs are 1. Due to the length of this trace, duplicate results are summarized.

```
decomp search:: inputs i=1 j=2
Found decomposition cdnes for function 1
        .
        . for all pairs of variables
        .
decomp search:: inputs i=4 j=5
Found decomposition cdnes for function 1
          ***chose: cdnes, inputs=1,2, outputs: f1=1, f2=1
decomp search:: inputs i=3 j=4
Found decomposition cdnes for function 1
        .
```

```
        . for all pairs of variables
        .
decomp search:: inputs i=5 j=7
Found decomposition cdnes for function 1
decomp search:: inputs i=6 j=7
          ***chose: cdnes, inputs=3,4, outputs: f1=1, f2=1
decomp search:: inputs i=5 j=6
Found decomposition cdnes for function 1
        .
        . for pairs of variables to i=7 j=8
        .
decomp search:: inputs i=7 j=8
Found decomposition cdnes for function 1
decomp search:: inputs i=7 j=9
Found decomposition sd00 for function 1
decomp search:: inputs i=8 j=9
          ***chose: sd00, inputs=7,9, outputs: f1=1, f2=0
decomp search:: inputs i=5 j=6
Found decomposition cdnes for function 1
        .
        . for all pairs of variables
        .
decomp search:: inputs i=8 j=10
Found decomposition cdnes for function 1
          ***chose: cdnes, inputs=5,6, outputs: f1=1, f2=1
decomp search:: inputs i=8 j=10
Found decomposition cdnes for function 1
decomp search:: inputs i=8 j=11
Found decomposition sd11 for function 1
decomp search:: inputs i=8 j=12
Found decomposition cdnes for function 1
decomp search:: inputs i=10 j=11
Found decomposition cdnes for function 1
decomp search:: inputs i=10 j=12
Found decomposition cdnes for function 1
decomp search:: inputs i=11 j=12
          ***chose: sd11, inputs=8,11, outputs: f1=1, f2=0
decomp search:: inputs i=10 j=12
Found decomposition cdnes for function 1
decomp search:: inputs i=10 j=13
Found decomposition cdnes for function 1
```

```
decomp search:: inputs i=12 j=13
Found decomposition cdnes for function 1
          ***chose: cdnes, inputs=10,12, outputs: f1=1, f2=1
decomp search:: inputs i=13 j=14
Found decomposition sdexor for function 1
decomp search:: inputs i=13 j=15
Found decomposition sd00 for function 1
decomp search:: inputs i=14 j=15
          ***chose: sdexor, inputs=13,14, outputs: f1=1, f2=0
decomp search:: inputs i=15 j=16
Found decomposition sd01 for function 1
          ***chose: sd01, inputs=15,16, outputs: f1=1, f2=0
        ***generated output 1 with line gate # 18
  18
   5
   6 OR      1   2   0
   7 AND     1   2   0
   8 OR      3   4   0
   9 AND     3   4   0
  10 OR      7   9   0
  11 OR      5   6   0
  12 AND     5   6   0
  13 AND     8  11   0
  14 OR     10  12   0
  15 AND    10  12   0
  16 XOR    13  14   0
  17 NOT    15   0
  18 AND    17  16   0
   0
# Circuit cost: Gates=  13  Lits=   25  Trans=   74  Levels=  5
# Total Prefilter costs: Gates=  0  Lits=  0  Trans=   0  Levels=  0
```

B.2 *adr2* - Sum of Two 2-bit Integers

Function *adr2* takes four inputs, composed of two 2-bit unsigned integer quantities and produces a 3-bit sum of these two values.

```
decomp search:: inputs i=1 j=2
```

```
decomp search:: inputs i=1 j=3
Found decomposition cdnes for function 1
Found decomposition sdexor for function 2
decomp search:: inputs i=1 j=4
decomp search:: inputs i=2 j=3
decomp search:: inputs i=2 j=4
Found decomposition sd11 for function 1
Found decomposition sd11 for function 2
Found decomposition sdexor for function 3
decomp search:: inputs i=3 j=4
           ***chose: sd11, inputs=2,4, outputs: f1=3, f2=0
           ***chose: sdexor, inputs=2,4, outputs: f1=4, f2=0
        ***generated output 3 with line gate # 6
decomp search:: inputs i=1 j=3
Found decomposition cdnes for function 1
Found decomposition sdexor for function 2
decomp search:: inputs i=1 j=5
Found decomposition cdnes for function 1
Found decomposition sdexor for function 2
decomp search:: inputs i=3 j=5
Found decomposition cdnes for function 1
Found decomposition sdexor for function 2
           ***chose: cdnes, inputs=1,3, outputs: f1=3, f2=1
decomp search:: inputs i=5 j=7
Found decomposition sd11 for function 1
Found decomposition sdexor for function 2
decomp search:: inputs i=5 j=8
Found decomposition sndx0 for function 1
decomp search:: inputs i=7 j=8
           ***chose: sd11, inputs=5,7, outputs: f1=1, f2=0
           ***chose: sdexor, inputs=5,7, outputs: f1=2, f2=0
        ***generated output 2 with line gate # 10
decomp search:: inputs i=8 j=9
Found decomposition sd00 for function 1
           ***chose: sd00, inputs=8,9, outputs: f1=1, f2=0
        ***generated output 1 with line gate # 11
M   11  10   6 0
   4
   5 AND     2   4   0
   6 XOR     2   4   0
   7 XOR     1   3   0
```

```
   8 AND      1   3   0
   9 AND      5   7   0
  10 XOR      5   7   0
  11 OR       8   9   0
   0
# Circuit cost: Gates=   7  Lits=   14  Trans=   42  Levels=  3
# Total Prefilter costs: Gates=  0  Lits=  0  Trans=   0  Levels=  0
```

B.3 *mlp2* - Product of Two 2-bit Integers

Function *mlp2* has four inputs and four outputs. The inputs are composed of two 2-bit unsigned integer quantities. The output is the product of these two quantities.

```
decomp search:: inputs i=1 j=2
Found decomposition sd11 for function 1
Found decomposition snd1x for function 2
decomp search:: inputs i=1 j=3
Found decomposition sd11 for function 1
Found decomposition sd11 for function 2
decomp search:: inputs i=1 j=4
Found decomposition sd11 for function 1
Found decomposition snd1x for function 2
Found decomposition sd11 for function 3
decomp search:: inputs i=2 j=3
Found decomposition sd11 for function 1
Found decomposition sndx1 for function 2
Found decomposition sd11 for function 3
decomp search:: inputs i=2 j=4
Found decomposition sd11 for function 1
Found decomposition sd11 for function 2
Found decomposition sd11 for function 4
decomp search:: inputs i=3 j=4
Found decomposition sd11 for function 1
Found decomposition snd1x for function 2
          ***chose: sd11, inputs=2,4, outputs: f1=b, f2=0
       ***generated output 4 with line gate # 5
decomp search:: inputs i=1 j=2
decomp search:: inputs i=1 j=3
```

```
Found decomposition sd11 for function 1
Found decomposition sd11 for function 2
decomp search:: inputs i=1 j=4
Found decomposition sd11 for function 3
decomp search:: inputs i=1 j=5
Found decomposition sd11 for function 1
Found decomposition sd10 for function 2
decomp search:: inputs i=2 j=3
Found decomposition sd11 for function 3
decomp search:: inputs i=2 j=4
decomp search:: inputs i=2 j=5
decomp search:: inputs i=3 j=4
decomp search:: inputs i=3 j=5
Found decomposition sd11 for function 1
Found decomposition sd10 for function 2
decomp search:: inputs i=4 j=5
          ***chose: sd11, inputs=1,3, outputs: f1=3, f2=0
          ***chose: sd11, inputs=1,4, outputs: f1=4, f2=0
decomp search:: inputs i=2 j=3
Found decomposition sd11 for function 3
decomp search:: inputs i=2 j=5
decomp search:: inputs i=2 j=6
decomp search:: inputs i=2 j=7
decomp search:: inputs i=3 j=5
decomp search:: inputs i=3 j=6
decomp search:: inputs i=3 j=7
decomp search:: inputs i=5 j=6
Found decomposition sd11 for function 1
Found decomposition sd01 for function 2
decomp search:: inputs i=5 j=7
decomp search:: inputs i=6 j=7
          ***chose: sd11, inputs=2,3, outputs: f1=4, f2=0
          ***chose: sd01, inputs=5,6, outputs: f1=2, f2=0
          ***chose: sd11, inputs=5,6, outputs: f1=1, f2=0
       ***generated output 1 with line gate # 11
       ***generated output 2 with line gate # 10
decomp search:: inputs i=7 j=8
Found decomposition sdexor for function 1
          ***chose: sdexor, inputs=7,8, outputs: f1=1, f2=0
       ***generated output 3 with line gate # 12
M   11  10  12   5 0
```

```
   4
   5 AND     2   4   0
   6 AND     1   3   0
   7 AND     1   4   0
   8 AND     2   3   0
   9 NOT     5   0
  10 AND     9   6   0
  11 AND     5   6   0
  12 XOR     7   8   0
   0
# Circuit cost: Gates=   8  Lits=   15  Trans=   44  Levels=  3
# Total Prefilter costs: Gates=  0  Lits=  0  Trans=   0  Levels=  0
```

B.4 *sqr3* - Square of 3-bit Integer

Function *sqr3* has three inputs and six outputs and produces the square of the 3-bit unsigned integer input.

```
decomp search:: inputs i=1 j=2
Found decomposition sd11 for function 1
Found decomposition sndlx for function 2
Found decomposition sdexor for function 3
decomp search:: inputs i=1 j=3
Found decomposition sndlx for function 2
Found decomposition sndxl for function 3
decomp search:: inputs i=2 j=3
Found decomposition sd10 for function 2
Found decomposition sndxl for function 3
Found decomposition sd10 for function 4
          ***chose: sd10, inputs=2,3, outputs: f1=a, f2=0
          ***chose: sdexor, inputs=1,2, outputs: f1=4, f2=0
        ***generated output 4 with line gate # 6
        ***generated output 6 with line gate # 3
decomp search:: inputs i=1 j=2
Found decomposition sd11 for function 1
decomp search:: inputs i=1 j=3
decomp search:: inputs i=1 j=6
Found decomposition sd10 for function 2
```

```
decomp search:: inputs i=1 j=7
decomp search:: inputs i=2 j=3
decomp search:: inputs i=2 j=6
decomp search:: inputs i=2 j=7
decomp search:: inputs i=3 j=6
decomp search:: inputs i=3 j=7
Found decomposition sd11 for function 3
decomp search:: inputs i=6 j=7
          ***chose: sd11, inputs=1,2, outputs: f1=1, f2=0
          ***chose: sd10, inputs=1,6, outputs: f1=2, f2=0
          ***chose: sd11, inputs=3,7, outputs: f1=4, f2=0
        ***generated output 1 with line gate # 8
        ***generated output 2 with line gate # 10
        ***generated output 3 with line gate # 11
M    8  10  11   6   4   3 0
   3
   4 GND     0
   5 NOT     3   0
   6 AND     2   5   0
   7 XOR     1   2   0
   8 AND     1   2   0
   9 NOT     6   0
  10 AND     1   9   0
  11 AND     3   7   0
   0
# Circuit cost: Gates=   8  Lits=   12  Trans=   34  Levels=  4
# Total Prefilter costs: Gates=  0  Lits=  0  Trans=   0  Levels=  0
```

Appendix C

Example Linearization Trace

This appendix provides a trace of a function that needs linearization to be realized by the two-place decomposition procedure. The function used is one of the 18 four-variable NPN-class representative functions identified by Miller [58] as having no decompositions.

```
decomp search:: inputs i=1 j=2
decomp search:: inputs i=1 j=3
decomp search:: inputs i=1 j=4
decomp search:: inputs i=2 j=3
decomp search:: inputs i=2 j=4
decomp search:: inputs i=3 j=4
Basis is:
c
a
5
b
Basis rows is:
d
a
5
3

*** In invert_mod2 ***
```

```
Input mat is:
d
a
5
3
After upper triangulation:
  mrows inv is:
13 8
7 12
2 14
1 15
Transform is:
a
d
e
f
Before mapping:
.i 4
.o 1
111- 1
0000 1
0011 1
-101 1
1-10 1
100- 0
-001 0
-100 0
011- 0
0-10 0
10-1 0
.e
After mapping:
.i 4
.o 1
1001 1
0110 1
0000 1
0001 1
0010 1
1000 1
0100 1
```

```
1001 1
1010 0
0101 0
1111 0
0101 0
1101 0
0111 0
0011 0
1100 0
1110 0
0011 0
0101 0
1011 0
.e
decomp search:: inputs i=7 j=6
decomp search:: inputs i=7 j=8
decomp search:: inputs i=7 j=9
Found decomposition sd00 for function 1
Allocating a new decomposition structure
decomp search:: inputs i=6 j=8
Found decomposition sd00 for function 1
Allocating a new decomposition structure
decomp search:: inputs i=6 j=9
decomp search:: inputs i=8 j=9
          ***chose: sd00, inputs=7,9, outputs: f1=1, f2=0
decomp search:: inputs i=6 j=8
Found decomposition sd00 for function 1
Allocating a new decomposition structure
decomp search:: inputs i=6 j=10
Found decomposition sndx1 for function 1
Allocating a new decomposition structure
decomp search:: inputs i=8 j=10
Found decomposition sndx1 for function 1
Allocating a new decomposition structure
          ***chose: sd00, inputs=6,8, outputs: f1=1, f2=0
decomp search:: inputs i=10 j=11
Found decomposition sd11 for function 1
Allocating a new decomposition structure
          ***chose: sd11, inputs=10,11, outputs: f1=1, f2=0
       ***generated output 1 with line gate # 13
  13
```

```
   4
   5 XOR     3   4   0
   6 XOR     5   2   0
   7 XOR     6   1   0
   8 XOR     5   1   0
   9 XOR     2   4   0
  10 OR      7   9   0
  11 OR      6   8   0
  12 AND    10  11   0
  13 NOT    12   0
   0
# Circuit cost: Gates=   9  Lits=   17  Trans=   50  Levels=  6
# Prefilter 1:
# Range from net 5 to net 9
# Cost: Gates=  5  Lits= 10  Trans=  30  Levels=  3
# Total Prefilter costs: Gates=  5  Lits= 10  Trans=  30  Levels=  3
```

Appendix D

CDNES Mapping Choice Examples

Here are presented two realizations generated by Decomp: one using the AND/OR mapping for CDNES decompositions and the other using the AND/XOR mapping.

The circuits are described using a net list description. The syntax of the file is as follows:

- The "#" character is the comment character. Everything to the right of it on a line is treated as a comment
- The first non-comment line specifies the numbers of the nets that are the function outputs. If there is only one output, the line consists of a single integer. If there are multiple outputs, the line consists of an "M" followed by a space delimited list of the output net numbers. The list is terminated by a "0".
- The second line contains a single integer indicating the number of inputs to the system of functions.
- The remaining lines describe the switching elements that are used, one per line.

The items specified are the output net number of the gate, the name of the gate and a list of gate input net numbers terminated by "0", in that order.

If a gate is an AOI or OAI, then the input groupings are specified within parentheses immediately preceding the gate input list.

The function that is used as an example is *9sym*.

The realization of *9sym* using the AND/OR mapping for the CDNES decomposition is:

```
 79
  9
 10 OR      1   2   0
 11 AND     1   2   0
 12 OR      3   4   0
 13 AND     3   4   0
 14 OR      5   6   0
 15 AND     5   6   0
 16 OR      7   8   0
 17 AND     7   8   0
 18 OR      9  10   0
 19 AND     9  10   0
 20 OR     11  12   0
 21 AND    11  12   0
 22 OR     13  14   0
 23 AND    13  14   0
 24 OR     15  16   0
 25 AND    15  16   0
 26 OR     17  18   0
 27 AND    17  18   0
 28 OR     23  26   0
 29 NOT    23   0
 30 AND    29  24   0
 31 NOT    30   0
 32 AND    20  31   0
 33 NOT    20   0
 34 AND    33  24   0
 35 NOT    25   0
```

```
36 AND    24  35   0
37 NOT    36   0
38 AND    22  37   0
39 NOT    22   0
40 AND    39  24   0
41 OR     21  40   0
42 NOT    40   0
43 AND    24  42   0
44 NOT    32   0
45 AND    28  44   0
46 NOT    19   0
47 AND    46  28   0
48 NOT    27   0
49 AND    48  28   0
50 XOR    28  43   0
51 OR     34  41   0
52 NOT    38   0
53 AND    52  47   0
54 NOT    47   0
55 AND    38  54   0
56 OR     45  49   0
57 AND    45  49   0
58 OR     55  56   0
59 OR     34  55   0
60 XOR    58  59   0
61 NOT    51   0
62 AND    61  60   0
63 NOT    60   0
64 AND    51  63   0
65 OR     50  62   0
66 AND    50  62   0
67 OR     53  57   0
68 AND    53  57   0
69 OR     34  67   0
70 OR     34  66   0
71 NOT    64   0
72 AND    71  69   0
73 NOT    68   0
74 AND    65  73   0
75 OR     65  72   0
76 XOR    70  75   0
```

```
  77 NOT    72   0
  78 AND    77  74   0
  79 OR     76  78   0
   0
# Circuit cost: Gates=  70  Lits=  123  Trans=  352  Levels= 18
# Total Prefilter costs: Gates=  0  Lits=  0  Trans=   0  Levels=  0
```

Contrast that with the following circuit generated using the AND/XOR mapping.

```
  39
   9
  10 XOR     1   2   0
  11 AND     1   2   0
  12 XOR     3   4   0
  13 AND     3   4   0
  14 XOR     5   6   0
  15 AND     5   6   0
  16 XOR     7   8   0
  17 AND     7   8   0
  18 XOR     9  10   0
  19 AND     9  10   0
  20 OR     11  19   0
  21 XOR    12  14   0
  22 AND    12  14   0
  23 OR     13  22   0
  24 XOR    15  17   0
  25 AND    15  17   0
  26 XOR    16  18   0
  27 AND    16  18   0
  28 OR     21  26   0
  29 XOR    20  23   0
  30 AND    20  23   0
  31 XOR    25  30   0
  32 XOR    24  27   0
  33 AND    24  27   0
  34 XOR    31  33   0
  35 XOR    28  29   0
  36 AND    28  29   0
  37 AND    32  35   0
  38 XOR    34  36   0
```

```
  39 XOR    37  38   0
   0
# Circuit cost: Gates=  30  Lits=   60  Trans=  180  Levels=  8
# Total Prefilter costs: Gates=  0  Lits=  0  Trans=   0  Levels=  0
```

Appendix E

Examples of Post-processing

Following are some examples showing the effects of post-processing.

E.1 2-out-of-5 Checker

Without post-processing the 2-out-of-5 Checker circuit is:

```
   5
   6 OR      1   2   0
   7 AND     1   2   0
   8 OR      3   4   0
   9 AND     3   4   0
  10 OR      7   9   0
  11 OR      5   6   0
  12 AND     5   6   0
  13 AND     8  11   0
  14 OR     10  12   0
  15 AND    10  12   0
  16 XOR    13  14   0
  17 NOT    15   0
  18 AND    17  16   0
   0
# Circuit cost: Gates=  13  Lits=   25  Trans=   74  Levels=  5
# Total Prefilter costs: Gates=  0  Lits=  0  Trans=   0  Levels=  0
```

With post-processing the circuit is:

```
  18
   5
   6 OR      1   2   0
   7 NAND    1   2   0
   9 NAND    3   4   0
  10 NAND    7   9   0
  12 AND     5   6   0
  13 OAI  (   2   2 )  3   4   5   6   0
  14 NOR    10  12   0
  16 XNOR   13  14   0
  18 AOI  (   2   1 ) 10  12  16   0
   0
# Circuit cost: Gates=   9  Lits=   21  Trans=   48  Levels= 5
# Total Prefilter costs: Gates=  0  Lits=  0  Trans=   0  Levels=  0
```

If XOR's are expensive in a given technology, then Decomp can be told not to generate XOR decompositions with the "-x" option. The circuit generated using "-x" is:

```
   5
   6 OR      1   2   0
   7 AND     1   2   0
   8 OR      3   4   0
   9 AND     3   4   0
  10 OR      7   9   0
  11 OR      5   6   0
  12 AND     5   6   0
  13 AND     8  11   0
  14 OR     10  12   0
  15 AND    10  12   0
  16 OR     13  15   0
  17 OR     14  16   0
  18 AND    14  16   0
  19 NOT    18   0
  20 AND    17  19   0
   0
# Circuit cost: Gates=  15  Lits=   29  Trans=   86  Levels= 7
# Total Prefilter costs: Gates=  0  Lits=  0  Trans=   0  Levels=  0
```

This can be reduced with "-r" as:

```
   5
   6 OR      1   2   0
   7 NAND    1   2   0
   9 NAND    3   4   0
  10 NAND    7   9   0
  12 AND     5   6   0
  13 OAI  (   2   2 )   3   4   5   6   0
  14 OR     10  12   0
  15 NAND   10  12   0
  16 NAND   13  15   0
  17 NOR    14  16   0
  20 AOI  (   2   1 )  14  16  17   0
   0
# Circuit cost: Gates=  11  Lits=   25  Trans=   56  Levels=  6
# Total Prefilter costs: Gates=  0  Lits=  0  Trans=   0  Levels=  0
```