# AUTOCORRELATION COEFFICIENTS IN THE REPRESENTATION AND CLASSIFICATION OF SWITCHING FUNCTIONS

by


JACQUELINE ELSIE RICE

M.Sc., University of Victoria, 1995
B.Sc., University of Victoria, 1993


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

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

---

Dr. J. C. Muzio, Co-Supervisor (Dept. of Computer Science)

---

Dr. M. Serra, Co-Supervisor (Dept. of Computer Science)

---

Dr. F. Ruskey, Committee Member (Dept. of Computer Science)

---

Dr. A. Gulliver, Outside Member (Dept. of Electrical & Computer Engineering)

---

Dr. M. Thornon, External Examiner (Dept. of Computer Science & Engineering, Southern Methodist University, Dallas Texas)

**Supervisors:** Dr. J. C. Muzio and Dr. M. Serra


## ABSTRACT

Reductions in the cost and size of integrated circuits are allowing more and more complex functions to be included in previously simple tools such as lawn-mowers, ovens, and thermostats. Because of this, the process of synthesizing such functions from their initial representation to an optimal VLSI implementation is rarely hand-performed; instead, automated synthesis and optimization tools are a necessity. The factors such tools must take into account are numerous, including area (size), power consumption, and timing factors, to name just a few. Existing tools have traditionally focused upon optimization of two-level representations. However, new technologies such as Field Programmable Gate Arrays (FPGAs) have generated additional interest in three-level representations and structures such as Kronecker Decision Diagrams (KDDs).

The reason for this is that when implementing a circuit on an FPGA, the cost of implementing exclusive-or logic is no more than that of traditional AND or OR gates. This dissertation investigates the use of the autocorrelation coefficients in logic synthesis for these types of structures; specifically, whether it is possible to pre-process a function to produce a subset of its autocorrelation coefficients and make use of this information in the choice of a three-level decomposition or of decomposition types within a KDD.

This research began as a general investigation into the properties of autocorrelation coefficients of switching functions. Much work has centered around the use of a function's spectral coefficients in logic synthesis; however, very little work has used a function's autocorrelation coefficients. Their use has been investigated in the areas of testing, optimization for Programmable Logic Arrays (PLAs), identification of types of complexity measures, and in various DD-related applications, but in a limited manner. This has likely been due to the complexity in their computation. in order to

investigate the uses of these coefficient, a fast computation technique was required, as well as knowledge of their basic properties. Both areas are detailed as part of this work, which demonstrates that it is feasible to quickly compute the autocorrelation coefficients.

With these investigations as a foundation we further apply the autocorrelation coefficient to the development of a classification technique. The autocorrelation classes are similar to the spectral classes, but provide significantly different information. This dissertation demonstrates that some of this information highlighted by the autocorrelation classes may allow for the identification of exclusive-or logic within the function or classes of functions.

In relation to this, a major contribution of this work involves the design and implementation of algorithms based on these results. The first of these algorithms is used to identify three-level decompositions for functions, and the second to determine decomposition type lists for KDD-representations. Each of these implementations compares well with existing tools, requiring on average less than one second to complete, and performing as well as the existing tools about 70% of the time.

**Examiners:**

Dr. J. C. Muzio, Co-Supervisor (Dept. of Computer Science)

Dr. M. Serra, Co-Supervisor (Dept. of Computer Science)

Dr. F. Ruskey, Committee Member (Dept. of Computer Science)

Dr. A. Gulliver, Outside Member (Dept. of Electrical & Computer Engineering)

Dr. M. Thornon, External Examiner (Dept. of Computer Science & Engineering, Southern Methodist University, Dallas Texas)

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In today's world, nearly every tool one uses is becoming computerized. This is primarily due to the reductions in the cost and size of computer chips. Because of these more and more complex functions can be incorporated into relatively simple tools such as thermostats, lawn-mowers, and ovens. One of the major problems inherent to these advances is the synthesis of the functions to be implemented in these tools. It is no longer possible to define, translate, and optimize many of these functions by hand due to their size and complexity. Automated synthesis and optimization programs are a necessity, and the factors that these programs must take into account are numerous. This dissertation addresses some of these issues, and introduces techniques that are of use in solving some of the known problems in the synthesis and optimization of switching functions.

The first issue in synthesis generally involves making a decision on the representation of the switching function to be implemented. Descriptions of switching functions range from textual to graphical, and may use only the Boolean domain or extend into the spectral domain. Every switching function $f(X) \in \{0, 1\}$, where $X = x_n, x_{n-1}, ..., x_2, x_1 \in \{0, 1\}$ must define the output for each of the $2^n$ possible input combinations. For even relatively small values of $n$, however, a textual representation listing each of these outputs is far too large to be of practical use. Many representations are still based on this concept, and use a variety of techniques to reduce the size of the list. All such representations tend to have the disadvantage of large size. Additionally, since the function is defined at a number of distinct points information about the overall structure of the function may be difficult to determine.

An alternative technique used to define switching functions expresses functions in terms of the Boolean operators used to combine the inputs. This may be a diagram of the circuit depicting the AND, OR, and various other gates, or it may be an expression such as a sum-of-products or product-of-sums. Tools such as Karnaugh maps may be used to convert from a truth table representation to one of these representations [1], which have the advantage of providing a better overall picture of the function. Again, though, for functions with large numbers of inputs, these representations may not be practical.

More recently, graph-based representations called decision diagrams (DDs) have been introduced [2]. DDs for most switching functions have the advantage of succintness, and are particularly useful in areas such as verification and testing [3] and in synthesis to field programmable gate array (FPGA) technologies [4]. One problem with DDs is that there are a variety of types, and so choosing the best type for a particular function is not an easy decision. Additionally, decisions related to the structure of the DD must be made as the graph structure is built. As is described in Chapter 2, an incorrect decision may mean the difference between a DD that is too large to store in memory and one that is relatively compact. This is an issue that we address in Chapter 7.

The above descriptions assume that the chosen representation is limited to the Boolean domain. If a translation to the spectral domain is performed, additional information about the function may become more readily apparent. This information may be used in the choice of one of the above representations, or in the process of synthesizing from one representation to another. There are various types of transformations, some of which have advantages over others. This research focuses primarily on the use of a representation that is based on the autocorrelation function.

The spectral and autocorrelation coefficients are alternative representations for switching functions. The advantages of these representations is that they are not limited to the Boolean domain, and thus display the information inherent within the function in a different way. It can be said that the spectral and autocorrelation coefficients of the function provide a more global view than do any of the above representations [5].

The spectral coefficients of a function are obtained by applying a transform matrix to the vector of the function's Boolean outputs. The resulting coefficients describe the switching function in terms of its similarity to the rows of the transform matrix. Re-applying the transform allows the regeneration of the original function.

The autocorrelation function provides a different type of transformation. The coefficients resulting from the application of this function describe the function in terms of its similarity to itself, shifted by a certain amount. This implies that the autocorrelation coefficients may be of great value in applications requiring knowledge about similarities within the switching function's structure. Chapter 2 provides background details for both the spectral transforms and the autocorrelation function.

Autocorrelation coefficients have previously been used in the areas of testing [6], optimization for Programmable Logic Arrays (PLAs) [7], identification of types of complexity measures [8], and in various DD-related applications [9, 10]. Their use has been relatively limited, however. This is most likely due to the complexity in the computation of the autocorrelation coefficients. As indicated in Chapter 5, this work includes the development of various techniques that overcome this problem.

Another approach to logic synthesis involves grouping switching functions into classes with some underlying similarities. There are $2^{2^n}$ possible Boolean functions of $n$ variables. Because of this there is a strong need to be able to group functions in some logical manner. One objective of classifying switching functions into such groupings is to list more compactly all $2^{2^n}$ possible functions. Another, more practical goal is to be able to state that certain information is true about all functions in a particular group, or, that all switching functions in a particular class have certain similarities or properties. From this a standard or canonical function for each class may be designated, thus leading to increased understanding about functions in that class, better fault diagnosis and testing procedures for that class of functions, and more efficient implementations [11]. It has been shown by Edwards [12] that it is possible to find a good implementation for a switching function by making use of an optimal implementation for the canonical representative of the function's class and then adding logic to the inputs or outputs as necessary.

This dissertation began as an investigation into the uses of the autocorrelation coefficients in logic synthesis and other digital logic applications. In order to make use of the coefficients, however, two other investigations were necessary: determining how to quickly compute the autocorrelation coefficients, and identifying their basic properties. This dissertation addresses the issue of computing the autocorrelation coefficients in Chapter 5, and determination of the properties of the autocorrelation coefficients is detailed in Chapter 3. This chapter presents properties of the autocorrelation coefficients such as limitations on the minimum and maximum values of the coefficients, the total values of the sum of the coefficients, and in general the values that the coefficients may take on. We also identify patterns within the autocorrelation coefficients that indicate the existence of exclusive-or (XOR) logic within the functions. Other patterns may be used to identify degenerate functions and sparse functions. All of these have uses in choosing function representations and for optimization and in minimizing the representations. Additional investigations into the use of the autocorrelation coefficients in the identification of symmetries were also carried out. These led to the discovery of a new type of symmetry that we label *anti-symmetries*. Details of this investigation are given in Chapter 4.

Since the complexity in their computation is no longer limiting, we propose to extend the applications of the autocorrelation coefficients to various areas of logic synthesis. One such area is that of classifying switching functions. Some of the motivation for this research has come from the fact that extensive work has been done in the area of spectral classification [12, 5, 11]; that is, classification based on a function's spectral coefficients. There are some overlaps and similarities between spectral and autocorrelation information, therefore many of the desirable properties of spectral classification are likely to be seen in a classification based on the autocorrelation coefficients. However, it is our hypothesis that the information in a function's autocorrelation coefficients is better suited to synthesis based on the newer representations such as DDs. The autocorrelation coefficients identify similarities within a function, which is exactly what a DD representation attempts to take advantage of. Chapter 6 presents our autocorrelation classes and includes a discussion on issues such as these.

Chapter 7 details the implementation of a tool for determining three-level logic decompositions that is based on some of the properties determined in Chapter 3. We compare the results with those of a known tool (AOXMIN-MV), and find that our autocorrelation-based tool agrees with AOXMIN-MV for 74% of the benchmarks and performs faster than AOXMIN-MV. Since our tool is limited to the identification of only two types of decompositions, this is a very promising result. In the same chapter the implementation and results of a tool for determining which type of decomposition to use at each level of a DD variant are presented. This tool does not perform as well as the three-level decomposition tool, but cannot be said to perform poorly. In comparison with a known sifting heuristic [13] our autocorrelation-based tool results in DDs with size within one node for 63% of the benchmarks. Both techniques have an average time of under one second, and the average number of nodes for each tool are within a difference of 1.5 Again, it is fair to say that for a simple algorithm based on this new work, these are results worth further investigation.

A more detailed outline of the dissertation is given below.

- Background material for this dissertation is presented in Chapter 2. We present an overview of switching functions and their representations, the spectral and autocorrelation coefficients, and classification techniques.
- Various properties of the autocorrelation coefficients are examined and proven

in Chapter 3. Minimum and maximum values are proven, as well as limitations on the sum of the coefficients and the values in general that the autocorrelation coefficients may take on. Theorems for the identification of sparse and degenerate functions are proven, as are theorems relating patterns in the autocorrelation coefficients to the existence of XOR logic within the function. Finally, some discussion is given on the potential relationship between autocorrelation coefficients of different orders.

- Chapter 4 examines the autocorrelation coefficients for their potential in identifying symmetries. Theorems relating patterns in a function's autocorrelation coefficients to the presence of symmetries within the function are proven, and details of the newly defined *anti-symmetries* are presented. Spectral conditions and tests for the anti-symmetries are derived, and applications of the anti-symmetries are discussed.
- Chapter 5 gives an overview of numerous techniques that have been developed and implemented for the computation of the autocorrelation coefficients, with an analysis of each. Two new computation techniques based on DDs are presented, and the results of experimental tests demonstrate that these techniques are the fastest for computation of coefficients for large benchmarks. A transform-based method is shown to be the most efficient for benchmarks with fewer than 10 inputs.
- We describe the autocorrelation classes in Chapter 6. These classes are based on our new classification technique that makes use of the autocorrelation coefficients. Four operations are defined as invariance operations for the autocorrelation classes, and canonical representatives of the classes are defined. Connections between the spectral and autocorrelation classes are discussed, and some potential uses of the autocorrelation classes are presented.
- Two applications of the properties investigated in Chapter 3 are detailed in

Chapter 7. The first application is in determining three-level decompositions, while the second is in determing decomposition types for a type of DD. Both tools are compared with existing tools for these applications with extremely promising results.

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

# Chapter 2

# Background

This chapter provides the background material required for the topics presented in this dissertation. Section 2.1 introduces the topic of switching (Boolean) functions, while Section 2.2 gives an overview of logic synthesis for switching functions and Section 2.3 discusses ways in which these functions can be represented, with emphasis on decision diagrams. The chapter also introduces a different domain for describing a switching function, the spectral domain. The spectral domain is introduced in Section 2.4 along with issues surrounding the computation of a function's spectrum, and uses of this representation in logic synthesis. Section 2.6 discusses ways in which Boolean functions can be classified, and the uses of this technique. The final section in the chapter introduces the autocorrelation function and other similar functions that may be applied to Boolean functions.

## 2.1 Switching Functions

Nearly all of today's logic systems are based on the Boolean-logic building blocks AND, OR, NAND and NOR. These operators were defined by George Boole in his Boolean algebra paper *The Calculus of Logic* [14]. The functions describing these logic systems are referred to as *Boolean functions* or *switching functions*, as the work by Boole was later applied to electronic switching circuits by C. E. Shannon [15]. This dissertation uses both terms interchangeably. Both the inputs and outputs of these functions are restricted to the Boolean domain of only two values, generally 0 and 1.

Switching functions have also been extended to allow more than two distinct values for both the inputs to the function and as the outputs of the function. This type of function is called a *multiple-valued function*, and is based on *multiple-valued logic* (MVL). This dissertation, however, concentrates only on switching functions in the two-valued Boolean domain over $\{0, 1\}$.

A Boolean function is a function $f(X)$ where

$$f(X) \in \{0, 1\}$$

$$X = \{x_n, x_{n-1}, ..., x_2, x_1\}$$

$$x_i \in \{0, 1\} \quad \forall \ i \quad \in \{1, 2, ..., n\}$$

The input vectors for the function can then take on $2^n$ possible values. If we assume that these input vectors are binary representations of a value $k$ such that

$$k = \sum_{i=1}^{n} x_i \cdot 2^{i-1}$$

then the input vectors can take on all possible values from 0 to $2^{n-1}$. The simplest way to illustrate this is to show a truth table for the function. For example, the truth table for the function $f(X) = x_1 \vee x_2$ is shown in Figure 2.1.

It should be noted that instead of restricting the domain of the inputs and output of a switching function to $\{0, 1\}$, instead the values $\{+1, -1\}$ may be used. This is expanded upon in Section 2.4.

The example in Figure 2.1 demonstrates one of the logical operations commonly used in Boolean functions, namely the logical OR ($\vee$) operator. Other operators are logical AND ($\wedge$), complementation ($\overline{x}$) and exclusive-OR ($\oplus$). The AND and OR operators are often denoted by $\cdot$ and $+$, respectively; however, the $\wedge$ and $\vee$ notation is used in this dissertation to avoid confusion with the more commonly known arithmetic

| $k$ | $x_2$ | $x_1$ | $f(X)$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 2 | 1 | 0 | 1 |
| 3 | 1 | 1 | 1 |

**Figure 2.1.** *The truth table for the function* $f(X) = x_1 \vee x_2$.

addition and multiplication operators. It should also be noted that it is common to leave out the operator when combining terms with either the multiplication operator or the AND operator. This usage is followed in this dissertation where the context clearly indicates which operator is intended.

The truth table for the OR operator is shown in Figure 2.1; Figure 2.2 demonstrates the functionality of the other operators. The result of the AND operation is

a)

| $x_2$ | $x_1$ | $x_1 \wedge x_2$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

b)

| $x_2$ | $x_1$ | $x_1 \oplus x_2$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

c)

| $x_1$ | $\overline{x}_1$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

**Figure 2.2.** *a) The truth table for the function* $f(X) = x_1 \wedge x_2$. *b) The truth table for the function* $f(X) = x_1 \oplus x_2$. *c) The truth table for the function* $f(X) = \overline{x}_1$.

called a *product* while the result of the OR operation is called a *sum*.

A Boolean function as shown in the example in Figure 2.1 is known as a *completely specified function*. That is, the output of the function is defined for all $2^n$ possible input combinations. It is also possible to have *incompletely specified functions*. These are Boolean functions in which the output values for some input combinations are not defined. These outputs are referred to as *don't cares* and are denoted with a dash (-). The analysis of incompletely specified functions is outside the scope of this work; all further discussions pertain only to completely specified functions.

Until now all of the functions illustrated have been functions with a single output. However, it is possible to have multiple outputs for a function. This is sometimes referred to as a system of functions, or an *m-output function*. Figure 2.3 shows a multiple-output function in which two of the output functions are incompletely specified. This dissertation considers only single-output completely specified functions.

| $x_3$ | $x_2$ | $x_1$ | $f_1(X)$ | $f_2(X)$ | $f_3(X)$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | – |
| 0 | 1 | 0 | 1 | – | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | – |
| 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 |

**Figure 2.3.** *A 3-variable 3-output incompletely specified function.*

## 2.2 Logic Synthesis

When used in reference to VLSI design, logic synthesis is most commonly defined as a two-step process consisting of [16]:

1. the optimization of a technology-independent logic representation, and

2. technology mapping.

The above steps are usually broken down into more detail as follows:

1. A standardized representation of the desired function is produced. Standard formats may vary from graphs such as binary decision diagrams (see Section 2.3) to equations describing the logic or languages such as Register Transfer Language (RTL).
2. The standard format is manipulated in order to minimize the logic, or to optimize with respect to some parameter(s) such as area and/or power consumption. This process generally consists of removing any redundancies and attempting to reduce the number of logic components.
3. Having reached a minimal or near minimal representation, the logic description must now be transformed to a format that is implemented in the desired technology. This format can vary from a list of basic gates to layouts that describe transistor structures.

Steps 1 and 2 are part of the technology-independent optimization phase, while step 3 is the technology-dependent step usually known as technology mapping.

**Step 1 - produce a representation of the function in a standard format** Languages such as VHDL[1] or RTL (Register Transfer Language) are often used to initially specify the function. In order to perform the next step of minimizing the logic, this description is often transformed into a two-level or multi-level representation of the function. A sum-of-products representation, as described in Section 2.3, is one example of a two-level representation.

**Step 2 - manipulate the function in order to minimize the logic** Depending on the representation chosen in step 1, either two-level or multi-level logic minimization is performed. When performing two-level minimization, the goal is

[1]VHDL stands for VHSIC Hardware Description Language. VHSIC stands for Very High Speed Integrated Circuits.

generally to find a minimal sum-of-products expression for the function. The objective of multi-level logic synthesis is to find the "best" multi-level structure, where "best" in this case means an equivalent representation that is optimal with respect to various parameters such as size, speed, or power consumption. Five basic operations are used in order to reach this goal:

i. Decomposition. This is the process of re-expressing a single function as a collection of new functions.

ii. Extraction. This is the process of identifying and creating some intermediate functions and variables, and re-expressing the original functions in terms of the intermediate plus the original variables. The process is used to identify the common sub-expressions.

iii. Factoring. This is the process of deriving a factored form from a sum-of-products form. The reason for this is to derive the minimum number of literals possible in the expression.

iv. Substitution. This is the process of expressing a function $F$ as a function of a second function, $G$, plus the original inputs to the function $F$. This is done by substituting $G$ into $F$ where ever possible.

v. Collapsing. This is also known as elimination, or flattening, and is the inverse of substitution.

These manipulations are repeated until the "best" structure (or close to it) is achieved. It is possible to use either *algebraic* or *Boolean* methods to perform the five operations listed above. Details and algorithms for both methods are given in [17].

**Step 3 - technology mapping**

Technology mapping is defined as a process of transforming a technology independent (optimized) Boolean network into a technology-based circuit [18]. Traditional techniques for technology mapping use a library of basic cells [19]. The Boolean network representing the circuit is transformed so that it uses only cells that exist in the library.

More recently logic synthesis, particularly the technology mapping phase, has had to take into account additional factors such as power consumption, physical size, timing constraints, and routing issues. The work in this dissertation has applications in both the technology-independent and dependent phases.

## 2.3 Representations of Switching Functions

The simplest way to represent a Boolean function is using its truth table, as shown in Section 2.1. A truth table is simply a table listing all possible inputs to the function along with the corresponding output(s). Clearly, for a completely specified function with $n$ variables, a truth table with $2^n$ rows is required to describe the function. This quickly becomes infeasible as the number of variables grows. Therefore there are many other ways to represent a Boolean function.

### 2.3.1 Karnaugh-maps

A map construction designed by Karnaugh [20] is commonly used for functions with small numbers of variables. A Karnaugh-map also shows all $2^n$ input combinations for a function; however, Karnaugh-maps have the advantage of reorganizing the information such that similar portions of the function may be grouped together. An example Karnaugh-map is shown in Figure 2.4. Each intersection of the rows and columns identifies the function for a particular assignment of the variables.

### 2.3.2 Sums, Products, and Related Representations

Another popular way to represent a switching function is as a *sum-of-products*, or *sop* expression. Additional notation is required to explain this. A *literal* is a variable $x_i$ or its complement $\overline{x}_i$. A product term is either a literal or a product of literals,

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 1 | 0 |
| 01 | 0 | 0 | 1 | 0 |
| 11 | 1 | 1 | 1 | 1 |
| 10 | 0 | 0 | 1 | 0 |

**Figure 2.4.** *The Karnaugh map for the function* $f(X) = x_4x_3 \vee x_2x_1$.

where a product of literals is a list of literals combined with the AND operator. A sum-of-products expression consists of a list of product terms combined with the OR operator.

The terms *minterm* and *maxterm* are also commonly used when discussing sop expressions. A product term in which each of the $n$ variables of a function appears exactly once in either its true or complemented form is called a *minterm*. A sum term is either a literal or a sum of literals, and a *maxterm* is a sum term in which each of the $n$ variables $x_i$ appears exactly once as either $x_i$ or $\overline{x_i}$.

The function shown in Figure 2.4 is expressed as a sum-of-products in the caption for the figure. There may be many different sum-of-product expressions for one function, so a *canonical form* is required for uniquely identifying the function. The canonical sum-of-products form of a function is a sum of minterms in which no two identical minterms appear, and it is created by summing the minterms for which $f(X) = 1$. A Karnaugh-map is a useful tool in minimizing the terms appearing in a sum-of-products representation of a function, as it allows groups of minterms for which $f(X) = 1$ to be easily identified.

Another way to express a Boolean function is as a *product-of-sums*. The canonical product-of-sums form is called a *maxterm expansion* and is a product of maxterms formed by multiplying the maxterms for which $f(X) = 0$ and in which no identical maxterms appear more than once.

### 2.3.3 Cube Lists

When using a function as an input to a synthesis tools it is common to use a *cube list*, or *cubic form* to describe the function. This notation involves representing non-canonical product terms by hyperplanes and edges of the cube defined by $2^n$ points in space. A three-variable example is shown in Figure 2.5.

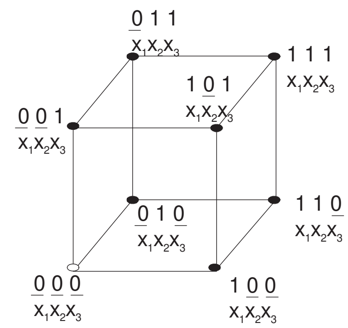


**Figure 2.5.** *A pictorial representation of the cubic form of the function* $f(X) = x_1 \vee x_2 \vee x_3$.

Based on Figure 2.5, one can describe the given function in a number of ways:

1. as a list of points:

   001, 010, 011, 100, 101, 110, 111

2. as a list of points, in terms of the variable values:

   $f(X) = \overline{x}_1\overline{x}_2x_3 \vee \overline{x}_1x_2\overline{x}_3 \vee \overline{x}_1x_2x_3 \vee x_1\overline{x}_2\overline{x}_3 \vee x_1\overline{x}_2x_3 \vee x_1x_2\overline{x}_3 \vee x_1x_2x_3$

3. as a list of planes for which all four points are in the *on-set*:

   $--1$

   $-1-$

   $1--$

4. as a list of planes, in terms of the variable values:

   $f(X) = x_1 \vee x_2 \vee x_3$

Items 2 and 4 are sum-of-products expressions for the same function. Items 1 and 3 are cube lists for the same function.

### 2.3.4 Decision Diagrams

A more recent representation for Boolean functions was popularized by Bryant [2]. This representation is called a Shannon tree, although it is sometimes referred to as an unreduced Binary Decision Diagram (BDD). An example Shannon tree is shown in Figure 2.6.

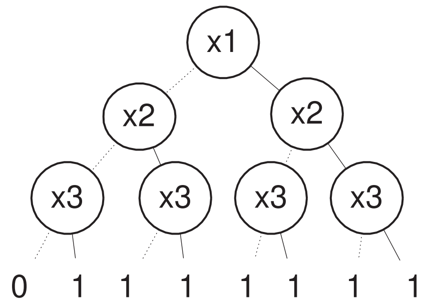


**Figure 2.6.** *The Shannon tree for the function $f = x_1 \vee x_2 \vee x_3$.*

A Shannon tree is defined as [21]

> *a binary directed acyclic graph with two leaves* TRUE *and* FALSE*, in which each non-leaf node is labeled with a variable and has two out-edges, one pointing to the subgraph that is evaluated if the node label evaluates to* TRUE *and the other pointing to the subgraph that is evaluated if the node label evaluates to* FALSE*.*

Every node in the tree represents either a literal in the Boolean function, or its complement. Every non-leaf node has two outward edges leading to two other nodes. If the node has a value of "1" (TRUE) then, to obtain the value of the expression, one follows the edge marked "1" and evaluates that node. Similarly, if the node has a value of "0" (FALSE), one follows the edge marked "0" and evaluates that node. This process is repeated until a leaf node with the value "1" or "0" is reached, and the evaluation is complete. The direction of the edges from each node is not explicitly marked, but is understood to be from the root towards the leaf nodes. In Figure 2.6 the 0 edge is the left edge leaving each node.

A Shannon tree makes use of the Shannon decomposition at each level of the graph. The Shannon decomposition of a function $f(X)$ is defined as [5]

$$f(X) = \overline{x}_n f_0 \vee x_n f_1$$

where

$$f_0 = f(0, x_{n-1}, ..., x_1)$$

and

$$f_1 = f(1, x_{n-1}, ..., x_1)$$

and, by relabeling the $x_i$ inputs, $x_n$ can be any of the original inputs. This structure had been introduced by Lee [22], and further described by Akers [23]. However, Bryant introduced algorithms for creating a canonical form of the structure called a *Reduced, Ordered* BDD (ROBDD). A ROBDD is a reduced BDD with a specified ordering of variables. A ROBDD meets two main specifications:

- a BDD is a reduced BDD if it contains no vertex whose left subgraph is equal to its right subgraph, nor does it contain distinct vertices $v$ and $v'$ such that the subgraph rooted by $v$ and $v'$ are isomorphic, and
- a BDD is an ordered BDD if on every path from the root node to an output, the variables are encountered at most once and in a specified order.

Figure 2.7 shows the ROBDD for the same function as depicted in Figure 2.6. The reduced BDD requires only 3 nodes not including the terminal or leaf nodes, while the unreduced BDD requires 7 nodes. Generally when the term BDD is used the writer is referring to a ROBDD. This practice is followed throughout this dissertation.

The addition of these specifications not only reduces the size required for storing functions, but it also ensures that the representation is canonical. This property, plus various implementation details and algorithms defined by Bryant [2] have combined to make this representation extremely efficient for operations such as evaluation, reduction, equivalence checking, satisfiability problems, and many others. The reader is directed to the vast amount of literature in this area, such as [24, 25, 4] for details.

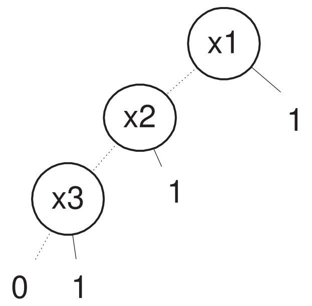


**Figure 2.7.** *The ROBDD for the function* $f = x_1 \vee x_2 \vee x_3$.

An additional technique for reducing the size of BDDs is the addition of *inverters*, also known as *complemented edges*. These are indicators on the path to a subgraph that are used to mark that the subgraph is inverted. This allows for the identification of even further isomorphic subgraphs within the BDD. However, inverters must be applied very carefully in order to maintain the property of canonicity. Rules surrounding their use are detailed in [25].

Another type of decision diagram called a *Functional Decision Diagram* (FDD) uses two different types of decomposition. The positive Davio decomposition for a function $f(X)$ is defined as

$$f(X) = f_0 \oplus x_n(f_0 \oplus f_1)$$

while the negative Davio decomposition is defined as

$$f(x) = f_1 \oplus \overline{x}_n(f_0 \oplus f_1).$$

If one allows all three types of decompositions to be used in one decision diagram, then it is generally referred to as a *Kronecker Functional Decision Diagram* (KFDD) or as the shortened form Kronecker Decision Diagram (KDD). Reduction and ordering rules similar to those described for BDDs are also applied to these two types of decision diagrams, although clearly the property of canonicity and the simplicity of implementation is more difficult to maintain.

Since Bryant's reductions for Binary Decision Diagrams were introduced many extensions and variants of this decision diagram structure have been developed. A few of these are described above; however, the reader is again directed to the many excellent references in this area for more complete descriptions of all the available types of decision diagrams.

## 2.4 The Spectral Domain

In Section 2.3 only representations limited to the Boolean domain were considered. The limitation of these representations is that only local information is provided; that is, at each input point the output is either a 1, a 0, or a don't care. If this restriction is lifted then it is possible to represent a Boolean function with a vector of values that each describe the function in a more global manner.

### 2.4.1 Spectral Transforms

In order to transform the function from the Boolean domain to what is called the *spectral domain*, some type of function or transform is applied. The resulting coefficients are called the *spectral coefficients* of the function. The vector of spectral coefficients is referred to as $R$ while the output vector of the function $f(X)$ is referred to as $Z$. The spectral transform is then computed as

$$R = T^n \times Z \tag{2.1}$$

### The Hadamard Transform Matrix

One of the commonly used spectral transforms is the Hadamard transform. It is defined as [5]

$$
\begin{gathered}
T^n \triangleq \begin{bmatrix} T^{n-1} & T^{n-1} \\ T^{n-1} & -T^{n-1} \end{bmatrix} \\
\text{where} \\
T^0 \triangleq [1]
\end{gathered}
\tag{2.2}
$$

An example of computing the spectral coefficients for a three-variable Boolean function is shown in Figure 2.8.

$$
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\
1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\
1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\
1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\
1 & -1 & -1 & 1 & -1 & 1 & 1 & -1
\end{bmatrix}
\times
\begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}
\begin{array}{c} z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{array}
=
\begin{bmatrix} 7 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \end{bmatrix}
\begin{array}{c} r_0 \\ r_1 \\ r_2 \\ r_{12} \\ r_3 \\ r_{13} \\ r_{23} \\ r_{123} \end{array}
$$

**Figure 2.8.** *Computing the spectral coefficients using the Hadamard transform matrix.*

It should be noted that this transform has some important properties.

- The transform matrix is of size $2^n \times 2^n$.
- The transform is reversible, that is, it is possible to compute the original values of $Z$ from $R$, since $Z = \frac{1}{2^n} T^n \times R$.
- Combining any two rows in the matrix using element-by-element multiplication results in a row that already is in the matrix.

### Other Transforms

Other transformations that are commonly used are the Walsh and Rademacher-Walsh. Definitions of these transforms are given in [5]. An example of the Walsh transform matrix for $n = 3$ is shown in Figure 2.9, while the Rademacher-Walsh transform matrix is shown in Figure 2.10. The spectral coefficients generated by either the

$$
\begin{array}{ccc}
k = & \begin{array}{cccccccc} 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \end{array} & \\
\begin{array}{r} j = 000 \\ 001 \\ 010 \\ 011 \\ 100 \\ 101 \\ 110 \\ 110 \end{array} &
\left[\begin{array}{rrrrrrrr}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\
1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\
1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \\
1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1
\end{array}\right] &
\begin{array}{l} Wal(0,k) \\ Wal(1,k) \\ Wal(2,k) \\ Wal(3,k) \\ Wal(4,k) \\ Wal(5,k) \\ Wal(6,k) \\ Wal(7,k) \end{array}
\end{array}
$$

**Figure 2.9.** *The Walsh transform matrix for $n = 3$.*

Walsh, Rademacher-Walsh, or the Hadamard transforms are the same, with the values appearing in different orderings. Each of these transforms also possess the properties described for the Hadamard transform.

Other transforms may also be used. [5] gives details of some of these other transforms.

## 2.4.2 The Meaning of the Spectral Coefficients

By multiplying the transform matrix by the output vector of the function the effect is to compare the Boolean function's output to the function represented by a given row of the transform. Figure 2.11 describes the functions against which the comparison

$$
\begin{array}{cl}
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\
1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\
1 & -1 & -1 & 1 & -1 & 1 & 1 & -1
\end{bmatrix}
&
\begin{array}{l}
Rad(0,k) \\
Rad(1,k) \\
Rad(2,k) \\
Rad(3,k) \\
Rad(1,k) \times (2,k) \\
Rad(1,k) \times (3,k) \\
Rad(2,k) \times (3,k) \\
Rad(1,k) \times (2,k) \times (3,k)
\end{array}
\end{array}
$$

**Figure 2.10.** *The Rademacher-Walsh transform matrix for* $n = 3$.

is being performed for the Hadamard transform matrix for $n = 3$.

It should be noted that in Figure 2.8 the function output vector is encoded as $\{0, 1\}$ while the functions in the transform matrix are encoded as $\{+1, -1\}$. However, it is common to re-encode the function output vector in $\{+1, -1\}$. In this case the output vector is referred to as $Y$, and the resulting spectral coefficients are referred to as $S$.

For each of the transforms discussed above, the resulting coefficients are labeled as shown in Figure 2.8. This labeling varies depending on the ordering of the rows in the transform matrix, and reflects the function comparison that is being performed with each row multiplication.

### 2.4.3 Properties of the Spectral Coefficients

As mentioned above, the advantage of describing a function using its spectral coefficients is that each coefficient provides a more global view of the function. In particular, there are a number of properties of the spectral coefficients. One property of the spectral coefficients is that if a function is an exact match with one of the rows of the transform then the resulting coefficient will have a maximum value, while the remaining coefficients are zero (if $\{+1, -1\}$ encoding is used). Additionally, a prop-

$$
\begin{array}{cl}
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
1 & 1 & -1 & -1 & 1 & 1 & 1 & 1 \\
1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 \\
1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\
1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\
1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\
1 & -1 & -1 & 1 & -1 & 1 & 1 & -1
\end{bmatrix}
&
\begin{array}{l}
Constant\ \ x_0 \\
x_1 \\
x_2 \\
x_1 \oplus x_2 \\
x_3 \\
x_1 \oplus x_3 \\
x_2 \oplus x_3 \\
x_1 \oplus x_2 \oplus x_3
\end{array}
\end{array}
$$

**Figure 2.11.** *The function represented by each row vector of the Hadamard transform matrix for $n = 3$.*

erty of the spectral transform is that no information is lost; that is, given the spectral coefficients and the transform used to generate them, it is possible to uniquely regenerate the original function. Finally, they can be used in classifying Boolean functions, as discussed in Section 2.6. Other properties and uses of the spectral coefficients are given in [5] and [26].

### 2.4.4 Computing the Spectral Coefficients

The computation of the $2^n$ spectral coefficients through application of Equation 2.1 (or the equivalent $S = T^n \times Y$) requires the summation of a total of $2^n \times 2^n$ individual product terms. For increasingly large values of $n$ this becomes infeasible. However, a faster method of performing the transform is possible since many subsets of intermediate values are common in the computation of the final coefficients. Figure 2.12 illustrates this.

The use of the fast transform reduces the number of terms to sum to $2^n \times n$. This is still very large for large values of $n$, but is considerably reduced from the orginal transform's requirements.

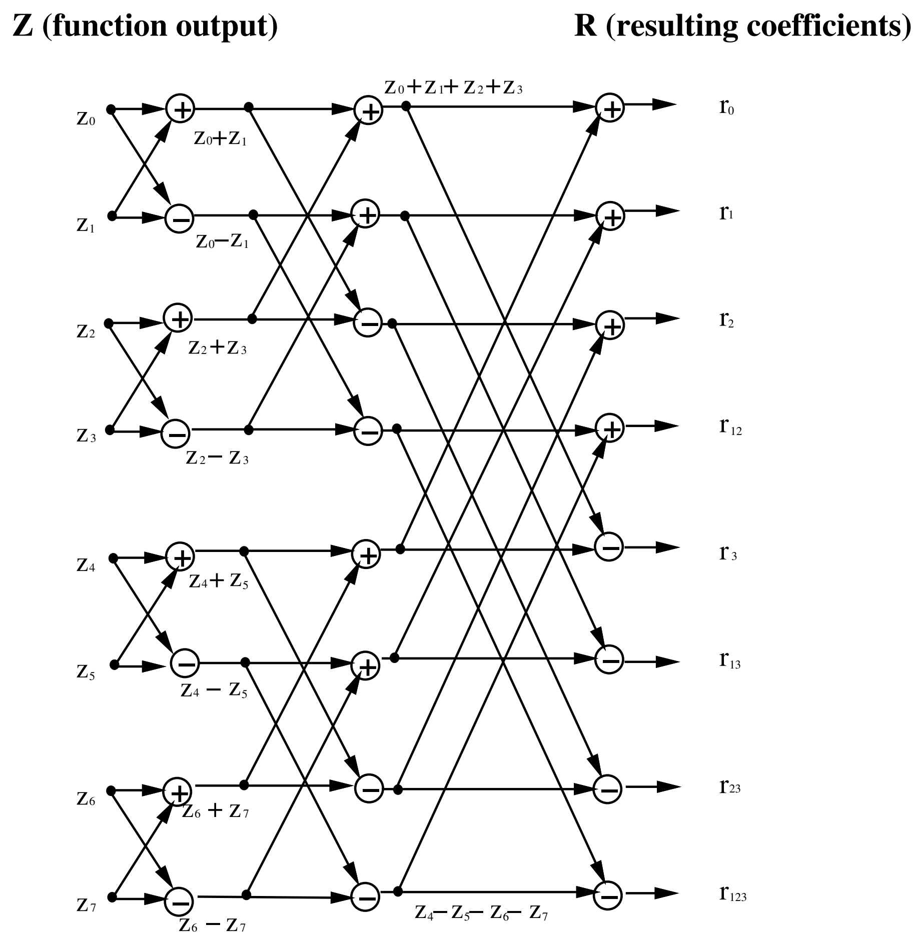


**Figure 2.12.** *A flow chart demonstrating the fast Hadamard transform for $n = 3$.*

Other methods of computing the spectral coefficient involve the use of BDDs, and are described in [27] and [28].

## 2.5 Autocorrelation

An alternate description of Boolean functions involves the use of correlation functions. As with the spectral transforms, the result is a vector of $2^n$ coefficients that describe the function.

### 2.5.1 Definition of the Correlation Function

The correlation function is defined as [26]

$$B^{f\,g}(\tau) = \sum_{v=0}^{2^n-1} f(v) \times g(v \oplus \tau) \tag{2.3}$$

where

$f$ and $g$ are both Boolean functions of $n$ or fewer variables using $\{0, 1\}$ encoding,

$v = \sum_{i=1}^{n} v_i \times 2^{i-1}$ while $v_i$ are the values assigned to each $x_i$, and

$\tau = \sum_{i=1}^{n} \tau_i \times 2^{i-1}$.

If $\{+1, -1\}$ encoding is used for the outputs of $f$ and $g$ then the resulting coefficient is labeled $C^{f\,g}(\tau)$.

### 2.5.2 Definition of the Autocorrelation Function

The autocorrelation function is defined identically to the correlation function, except that both functions involved are the same:

$$B^{f\,f}(\tau) = \sum_{v=1}^{2^n-1} f(v) \times f(v \oplus \tau) \tag{2.4}$$

In general, the superscript $f\ f$ is omitted when referring to the autocorrelation function. For the remainder of this dissertation, $B$ ($C$) is used to refer to the entire vector of autocorrelation coefficients, and $B(\tau)$ ($C(\tau)$) is used to refer to each entry in this vector.

Techniques for computing the autocorrelation function are described in detail in Chapter 5.

### 2.5.3 Meaning and Labeling of the Autocorrelation Coefficients

Figure 2.13 shows an example of computing the autocorrelation coefficients for a small function.

$$B(\tau) = \sum_{v=1}^{2^n-1} f(v) \times f(v \oplus \tau)$$

$$\begin{bmatrix} B(0) \\ B(1) \\ B(2) \\ B(3) \\ B(4) \\ B(5) \\ B(6) \\ B(7) \end{bmatrix} = \begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{bmatrix} \begin{bmatrix} f(0 \oplus 0) & f(1 \oplus 0) & \dots & f(7 \oplus 0) \\ f(0 \oplus 1) & f(1 \oplus 1) & \dots & f(7 \oplus 1) \\ f(0 \oplus 2) & f(1 \oplus 2) & \dots & f(7 \oplus 2) \\ f(0 \oplus 3) & f(1 \oplus 3) & \dots & f(7 \oplus 3) \\ f(0 \oplus 4) & f(1 \oplus 4) & \dots & f(7 \oplus 4) \\ f(0 \oplus 5) & f(1 \oplus 5) & \dots & f(7 \oplus 5) \\ f(0 \oplus 6) & f(1 \oplus 6) & \dots & f(7 \oplus 6) \\ f(0 \oplus 7) & f(1 \oplus 7) & \dots & f(7 \oplus 7) \end{bmatrix} = \begin{bmatrix} 7 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \end{bmatrix}$$

**Figure 2.13.** *An example of computing the autocorrelation coefficients for* $f(X) = x_1 \vee x_2 \vee x_3$.

It is also common to label the entries of the autocorrelation vector with the binary encoding of $\tau$. The computation can then be written, for example for $\tau = 3$ as

$$B(011) = f(000) \times f(000 \oplus 011) + \ldots + f(111) \times f(111 \oplus 011)$$

This makes the meaning of the autocorrelation coefficients much more evident. One can now see that with each XOR operation, certain input bits are inverted. Only the input bits in positions corresponding to 1's in the binary expansion of $\tau$ are inverted. As noted earlier the ordering of the input variables is $x_n \ldots x_1$. Thus another relabeling of the entries of the autocorrelation vector is to indicate which of the input bits are being inverted in the XOR operations. Figure 2.14 demonstrates each of these alternative labelings.

The autocorrelation coefficients are generally divided into groupings according to the number of 1's in the value of $\tau$. Thus $B(000)$ is the zero-order autocorrelation coefficient, and $B(001)$ is a first order coefficient, as is $B(010)$ and $B(100)$. $B(011)$,

$$\begin{bmatrix} B(0) \\ B(1) \\ B(2) \\ B(3) \\ B(4) \\ B(5) \\ B(6) \\ B(7) \end{bmatrix} = \begin{bmatrix} B(000) \\ B(001) \\ B(010) \\ B(011) \\ B(100) \\ B(101) \\ B(110) \\ B(111) \end{bmatrix} = \begin{bmatrix} B(0) \\ B(x_1) \\ B(x_2) \\ B(x_1x_2) \\ B(x_3) \\ B(x_1x_3) \\ B(x_2x_3) \\ B(x_1x_2x_3) \end{bmatrix}$$

**Figure 2.14.** *Alternative labelings for the autocorrelation coefficients (assuming $n = 3$).*

$B(101)$ and $B(110)$ are second order coefficients, and so on. More formally, the order of a coefficient $B(\tau)$ is defined as the weight $|\tau|$, or the number of ones in the binary expansion of $\tau$.

### 2.5.4 Related Concepts

There are a number of concepts with uses in digital logic that are closely related to the autocorrelation coefficients.

#### The Boolean Difference

The Boolean difference of a Boolean function is a computation that has been used to evaluate test patterns for digital circuits, and also has applications in logic synthesis. It is defined as [29]

$$\frac{df(X)}{dx_i} = f(x_n, \ldots, x_i, \ldots x_1) \oplus f(x_n, \ldots, \overline{x}_i, \ldots x_1). \tag{2.5}$$

The Boolean difference results in a function that evaluates to 0 if there is no change to the function's output from inputs containing $x_i$ to inputs containing $\overline{x}_i$, and 1 otherwise. By identifying the number of input combinations that result in the function $\frac{df(X)}{dx_i}$ having the value 0 and subtracting the number that result in the value 1, the result is the same as the corresponding autocorrelation coefficient $C(\tau)$, where $\tau$ is a binary value with one 1 in position $i$.

### The Gibbs Differentiator

The partial Gibbs derivative of a function $f$ with respect to a variable $x_i$ is defined by Stanković [30] as

**Definition 2.6**

$$D_i\{f(X)\} = f(x_n, \ldots, \overline{x}_i, \ldots, x_1) - f(x_n, \ldots, x_i, \ldots, x_1). \tag{2.6}$$

This is referred to as the *partial* derivative because it is computed with respect to only one of the inputs, $x_i$. The Gibbs derivative of a function is defined, again from Stanković [30], as

$$D\{f(X)\} = -\frac{1}{2}\sum_{i=1}^{n} 2^{i-2}(D_i\{f(X)\}) \tag{2.7}$$

where $n$ is the number of variables in the function. Stanković uses $\{0, 1\}$ encoding of the function $f$ for the above definitions, while Edwards [31] uses $\{+1, -1\}$ encoding of the function $f$ and defines this derivative as

$$D\{f(X)\} = \sum_{i=1}^{n} 2^{i-1}\{f(x_n, \ldots, x_i, \ldots, x_1) - f(x_n, \ldots, \overline{x}_i, \ldots, x_1)\} \tag{2.8}$$

It should be noted that the order of the subtraction for Edwards' and Stanković's definitions is reversed. In both definitions, the difference of the function's value at one point is found when compared to another point, and then these differences are summed.

From the definitions above one can infer the meaning that the partial Gibbs derivative is the difference between the function at one input point and at another input point a unit vector away in direction $x_i$. For example, if there is no change in the function from, say, $x_2x_1 = 00$ to $x_2x_1 = 01$ then the value of the partial derivative with respect to $x_1$ at $x_2x_1 = 00$ is 0.

When these partial values are summed and a weighting factor is incorporated, as in the definition for the total Gibbs derivative, then the value of the derivative has a most significant bit which indicates the slope of the function in direction $x_1$ away from the starting point, a next significant bit which indicates the slope of the function in direction $x_2$, and so on. This is most easily demonstrated using a Karnaugh map to represent the function.

| $x_2x_1 \backslash x_3$ | 0 | 1 |
|---|---|---|
| 00 | 0 | 1 |
| 01 | 0 | 1 |
| 11 | 1 | 1 |
| 10 | 0 | 1 |

| $x_2x_1 \backslash x_3$ | 0 | 1 |
|---|---|---|
| 00 | 1 | $-1$ |
| 01 | 1 | $-1$ |
| 11 | $-1$ | $-1$ |
| 10 | 1 | $-1$ |

**Figure 2.15.** *The Karnaugh map for the function $f(X) = x_1x_2 \vee x_3$, showing $\{0,1\}$ encoding of the outputs on the left and $\{+1,-1\}$ encoding of the outputs on the right.*

The value of $D\{f(110)\}$ can be computed two ways: using Stanković's [30] definition (see Definition 2.7) or Edwards' [31] definition (see Definition 2.8). Using Edwards' definition, the computation is

$$\begin{aligned} D\{f(011)\} &= 2^1 \times \{f(011) - f(111)\} + \\ &\quad 2^0 \times \{f(011) - f(001)\} + \\ &\quad 2^{-1} \times \{f(011) - f(010)\} \\ &= -3 \end{aligned}$$

The binary representation of 3 is 011, indicating a slope of 1 in the $x_1$ direction, a slope of 1 in the $x_2$ direction, and a slope of 0 in the $x_3$ direction. Each of these directions are indicated with arrows in the Karnaugh map in Figure 2.16.

| $x_2x_1 \backslash x_3$ | 0 | 1 |
|---|---|---|
| 00 | 1 | $-1$ |
| 01 | 1 | $-1$ |
| 11 | | $-1$ |
| 10 | 1 | $-1$ |

**Figure 2.16.** *The Karnaugh map for the function* $f(X) = x_1x_2 \vee x_3$, *showing* $D\{f(110)\}$.

The Gibbs differential could potentially be used to compute the first order autocorrelation coefficients. An example is shown below. By summing $2^n - 1$ $D$ values, the number of places where $f(x_3, x_2, x_1)$ disagrees with $f(\overline{x}_3, x_2, x_1)$ can be computed. It is possible to calculate $B(\tau)$ by subtracting this value from $B(0)$, the zero-order coefficient.

$$B(0) - B(100) = D\{f(000)\} + D\{f(010)\} + D\{f(001)\} + D\{f(011)\}$$

$$B(0) - B(x_n \ldots 1 \ldots x_1) = \sum_{x_i \ldots x_{i-1}, x_{i+1} \ldots x_n = 0}^{2^n - 1} D\{f(x_n \ldots 0 \ldots x_1)\}$$

### Complexity Measure

It can be quite difficult to examine a Boolean function and provide some measure of its complexity. One measure that has been suggested is the following [5][2]:

$$Cmp(f) = n2^n - \frac{1}{2^{n-2}}\{\sum_{v=0}^{2^n-1} \parallel v \parallel r_v^2\} \tag{2.9}$$

where $\parallel v \parallel$ is a weighting factor[3] representing the order of the spectral coefficient $r_v$. The term $n2^n$ represents the total number of adjacent minterms in $f(X)$, where

[2] The standard notation for the complexity measure of a Boolean function is $C(f)$; we use $Cmp(f)$ to avoid confusion with the $\{+1, -1\}$ autocorrelation coefficient $C(\tau)$.

[3] $\parallel v \parallel = 0$ for $r_0$, $\parallel v \parallel = 1$ for $r_1$ to $r_n$, $\parallel v \parallel = 2$ for $r_{12}$ to $r_{n-1n}$, and so on.

| $x_3x_4 \backslash x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 0 | 1 |
| 01 | 1 | 0 | 1 | 0 |
| 11 | 0 | 1 | 0 | 1 |
| 10 | 1 | 0 | 1 | 0 |

$(a)$

| $x_3x_4 \backslash x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 1 | 1 | 1 | 0 |
| 11 | 1 | 1 | 1 | 0 |
| 10 | 0 | 1 | 1 | 0 |

$(b)$

**Figure 2.17.** *Example functions illustrating the complexity measure* $Cmp(f)$*; (a)* $f(X) = x_1 \oplus x_2 \oplus x_3 \oplus x_4$ *has* $Cmp(f) = 0$*; (b)* $f(X) = \overline{x}_1 x_4 \vee x_2(x_3 \vee x_4)$ *has* $Cmp(f) = 40$.

adjacent minterms are minterms at a Hamming distance of one from each other. Each pair is counted twice. The term $\frac{1}{2^{n-2}}$ represents the number of different-valued minterm pairs; *i.e.* pairs in which one minterm evaluates to 0 and one minterm evaluates to 1. An example is shown in Figure 2.17.

The autocorrelation coefficients can also be used in this computation [32]:

$$Cmp(f) = \sum_{\|u\|=1} B(u) + (2^n - B(u)) - 2(B(0) - B(u))$$

$\| u \| = 1$ means that the computation is summed only over the first-order autocorrelation coefficients. The expression $B(u)$ gives the number of places where the pair of minterms at a Hamming distance of one are both 1's. The expression $2^n - B(u)$ gives the number of places where the pair of minterms are not both 1's; *i.e.* 0-1 pairs and 0-0 pairs, and from that the expression $2(B(0) - B(u))$, giving the 0-1 pairs, is subtracted. The result is the number of 1-1 pairs plus the number of 0-0 pairs each at a Hamming distance of one, which is the complexity measure.

## 2.6 Classification of Switching Functions

There are $2^{2^n}$ possible Boolean functions of $n$ variables. Even for very small values of $n$ this is a number that is far too large for any systematic evaluation of every possible function. It is for this reason that methods of classifying Boolean functions into groups with known characteristics are useful.

### 2.6.1 NPN Classification

One of the most common ways to classify Boolean functions is to use some simple operations, namely permutation and negation. A function $f$ is said to be *NPN-equivalent* to another function $g$ if it is possible to convert $f$ into $g$ by applying one or more of the following transformations to $f$.

a) **N**egation of one or more of the input variables,

b) **P**ermutation of two or more of the input variables, or

c) **N**egation of the output of the function.

Below are four functions that are NPN-equivalent.

$$\begin{aligned} f_1(X) &= x_1\overline{x}_2 \vee x_2x_3 \\ f_2(X) &= x_1x_2 \vee \overline{x}_2x_3 \\ f_3(X) &= x_1x_3 \vee x_2\overline{x}_3 \\ f_4(X) &= \overline{x}_1x_3 \vee \overline{x}_2\overline{x}_3. \end{aligned}$$

$f_2(X)$ is equivalent to $f_1(X)$ when $x_1$ and $x_3$ are permuted. $f_3(X)$ is equivalent to $f_1(X)$ through the negation of $x_2$, and $f_4(X)$ is $f_3(X)$ negated. Functions that are NPN-equivalent are said to be in the same *NPN-equivalence class*. Smaller classes can also be considered by using only two or one of these operations; this creates *P-equivalence* classes (operation b only), *N-equivalence* classes (operation a only), and *NP-equivalence* classes (operations a and b only). The number of equivalence classes for functions of up to five variables has been extensively discussed, but is generally only tabulated for $n \leq 3$ [5, 11].

According to Hurst, the total number of possible functions of less than or equal to 4 variables can be compressed into 402 NP-equivalence classes. This can be further compressed into 222 NPN-equivalence classes.

### 2.6.2 Threshold Functions

We briefly mention linearly separable functions, or threshold functions. These are functions for which, when their minterms are considered as $2^n$ equispaced nodes in $n$-dimensional space, there exists a plane that separates all the true ($f(X) = 1$) nodes from all the false nodes ($f(X) = 0$). An example is shown in Figure 2.18. This

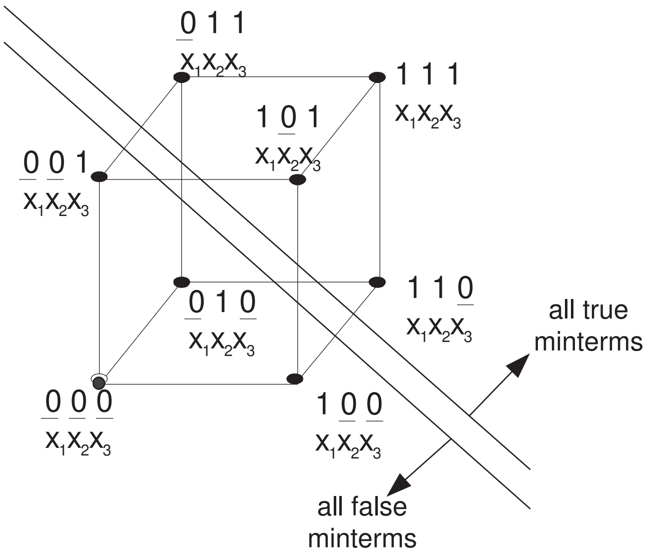


**Figure 2.18.** *An illustration of the concept of linearly separable, or threshold, functions.*

is a useful property, as a function with this property may be implemented using a threshold gate, this being a gate which switches when a given threshold of inputs is reached. It should be noted that of the $2^{2^n}$ possible Boolean functions of $n$ variables, a decreasingly small percentage of them are linearly separable (as $n$ increases); however, many functions that are used in everyday applications, such as arithmetic and decision mechanisms, are linearly separable [5].

The need to identify linearly separable functions led to another method of classification that uses $n+1$ parameters to describe each class of functions. These parameters are called Chow parameters, and they are defined as:

$$\begin{aligned}
&CH(X) \triangleq Ch(x_1), Ch(x_2), ..., Ch(x_n); Ch(x_0) \\
&\text{where} \\
&Ch(x_i) = \sum occurrence\ \ of\ \ x_i\ \ over\ \ all\ \ true\ \ minterms;\ \ i = 1\ \ to\ \ n \\
&Ch(x_0) = total\ \ number\ \ of\ \ true\ \ minterms
\end{aligned} \tag{2.10}$$

The Chow parameters may be used to characterize classes which encompass the NPN classes. The main use of this classification is to identify whether or not a function is linearly separable [5].

### 2.6.3 Spectral Classification

A modified version of the Chow parameters are actually the first order spectral coefficients. The spectral coefficients are also used extensively in classification of Boolean functions. If the entire vector of spectral coefficients for a function is considered, there are five invariance operations that may be identified. Invariance operations are operations that leave the magnitudes of the individual coefficients unchanged although the order or sign of the individual values may change. These invariance operations are listed below [5].

(i) Permutation of two variables $x_i$ and $x_j$, $i \neq j$ and $i, j \in 1..n$. The result of this is the interchange of $2^{n-2}$ pairs of spectral coefficients,

$$\begin{gathered}
s_i \leftrightarrow s_j \\
s_{ik} \leftrightarrow s_{jk} \\
\vdots
\end{gathered}$$

and $s_0, s_k, s_{ij}, ...$ are unchanged.

(ii) Negation of one variable $x_i$. The result of this is the negation of $2^{n-1}$ spectral coefficients,

$$s_i \rightarrow -s_i$$
$$s_{ij} \rightarrow -s_{ij}$$
$$\vdots$$

and $s_0, s_j, s_{j\alpha}$ are unchanged. $\alpha$ represents all or some of the remaining variables, not including $i$ or $j$.

(iii) Negation of the output. The result of this is that all the spectral coefficients are negated.

$$s_0 \rightarrow -s_0$$
$$s_i \rightarrow -s_i$$
$$\vdots$$

(iv) Replacement of $x_i$ with $x_i \oplus x_j$. The result of this is that $2^{n-2}$ pairs of coefficients are interchanged,

$$s_i \leftrightarrow s_{ij}$$
$$s_{i\alpha} \leftrightarrow s_{ij\alpha}$$

and $s_0, s_j, s_{j\alpha}$ are unchanged.

(v) Replacement of the output $f(X)$ with $f(X) \oplus x_i$. The result of this is the interchange of $2^{n-1}$ pairs of spectral coefficients; in other words, all coefficients are modified.

$$s_i \leftrightarrow s_0$$
$$s_{ij} \leftrightarrow s_j$$
$$s_{ij\alpha} \leftrightarrow s_{j\alpha}$$

These operations in combination allow any spectral coefficient value to be moved to any other position in the coefficient vector. The magnitudes of the $2^n$ coefficients therefore provide a classification technique that encompasses both the NPN classes and the Chow classification. In fact, there are 8 classes of functions where $n \leq 4$. This leads to a very compact classification in which the 65536 functions for $n \leq 4$ are divided into only 8 classes. Clearly, analysis of 8 representative functions is far more feasible than analysis of 65536 functions.

### 2.6.4 Other Classification Techniques

There are a variety of other classification techniques which are not relevant to this dissertation. However, should the reader wish more information in this area some context is given in Figure 2.19. The reader is directed to [5] for further details on

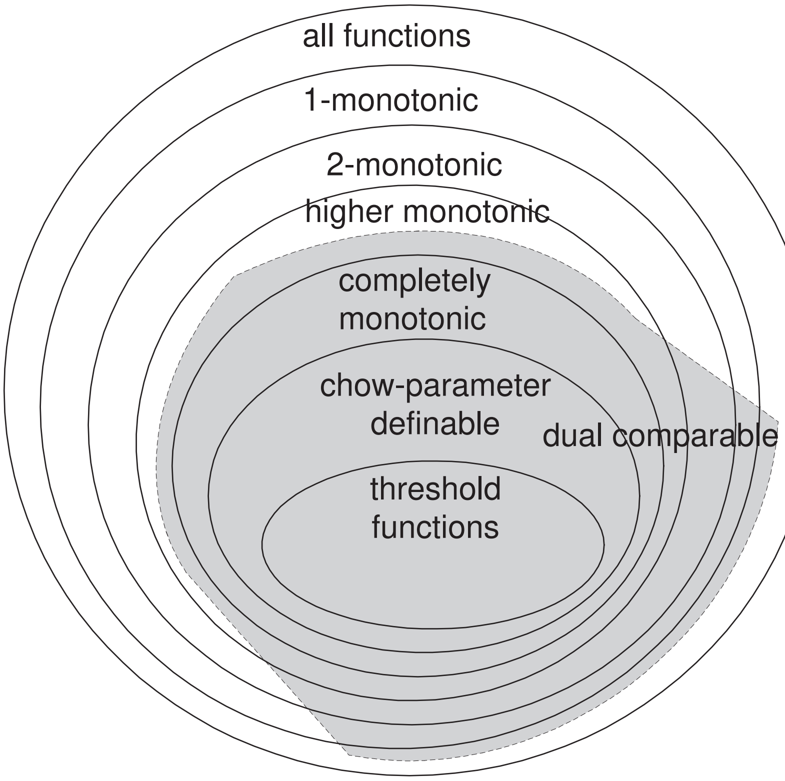


**Figure 2.19.** *A diagram illustrating how various classes of functions are related.*

classifications such as unateness, monotonicity, summability and dual comparability, and to [33] for details on a more recent classification technique that has been proposed based on fixed polarity Reed-Müller forms.

### 2.6.5 Use of Classification

One of the problems in the design of logic circuits is to synthesize a circuit implementation from a function description. Classification of functions has provided one possible solution to the question of how to efficiently perform this task. By making use of the spectral technique for classification one can identify functions within the same spectral class. Time and effort can then be spent to find the best possible implementation for a canonical function from the class. By adding logic to the inputs or outputs of this function one can relatively quickly and efficiently implement any of the other functions in the spectral class, as shown by Edwards [12]. This is particularly useful if there are known characteristics for the canonical function that make it "easy" to implement, such as the presence of symmetries in the function.

Another goal of classification has been to identify a universal logic element, or, a single function from which all $2^{2^n}$ functions of a given $n$ could be realized. More information on this application of classification can be found in [12, 5] and [11].

## 2.7 Symmetries

Another useful property to identify in switching functions is symmetry. Partial symmetries exist in most Boolean functions, particularly those used in practical applications. Both total and partial symmetry properties are commonly used in synthesis of digital circuits [5, 34, 35, 36], particularly in the reduction of the size of Binary Decision Diagram (BDD) representation of functions [37, 38].

A function is symmetric in some way if two or more of the variables can be interchanged leaving the output of the function unchanged. The sections below describe various types of symmetries.

### 2.7.1 Totally Symmetric Functions

A Boolean function is totally symmetric if the output is unchanged by any permutation of the inputs to the function. For example, the majority function $f(X) = x_1 \vee x_2 \vee x_3$ is totally symmetric, as is the function $f(X) = x_1x_2 \vee x_2x_3 \vee x_1x_3$.

### 2.7.2 Symmetric Functions

There are in practice very few totally symmetric functions, and they form a decreasing percentage of the total number of Boolean functions ( $\frac{2^{n+1}}{2^{2^n}}$ ). For this reason, a less restrictive definition of symmetry is more useful.

**Definition 2.1** *A Boolean function is* symmetric *in two or more input variables* $\{x_i, \ldots, x_k\}$ *if any interchanges within this nonempty subset leave* $f(X)$ *unchanged [5].*

For example, $f(X) = x_1\overline{x}_3 \vee x_2\overline{x}_3 \vee x_1x_2x_4$ is symmetric in $\{x_1, x_2\}$, since permuting these two variables does not alter the outputs of the function. A totally symmetric function may be thought of as a special case of symmetric functions.

In this work we are interested in identifying any portion of the function for which any interchange of two variables or their inverses does not alter the outputs of the function. This is known as a symmetry of degree 2. There are various types of symmetries of degree 2 which are described in the following section.

### 2.7.3 Symmetries of Degree 2

If a function has a symmetry that can be identified by assigning two variables a set of fixed values then we call this a symmetry of degree 2. In this case the degree refers to the number of variables being examined.

Symmetries of degree 2 can be identified by finding patterns where

$$f(x_n, \ldots, a, \ldots, b, \ldots, x_1) = f(x_n, \ldots, c, \ldots, d, \ldots, x_1),$$

$a, b, c, d \in \{0, 1\}$. The easiest way to illustrate these properties is by examining the function's Karnaugh map.

### Equivalence Symmetries

One type of symmetry of degree 2 is referred to as equivalence symmetries. These are defined as follows: a function $f(x_n, \ldots, x_i, \ldots, x_j, \ldots, x_1)$ has an equivalence symmetry in $\{x_i, x_j\}$ if

$$f(x_n, \ldots, 0, \ldots, 0, \ldots, x_1) = f(x_n, \ldots, 1, \ldots, 1, \ldots, x_1) \tag{2.11}$$

In other words, if the function's values for $x_ix_j = 00$ are unchanged when both $x_ix_j = 11$ then the function has an equivalence symmetry in these variables. This is written $E\{x_i, x_j\}$. For example, the function $f(x_4, x_3, x_2, x_1) = \overline{x}_1\overline{x}_2\overline{x}_3 \vee \overline{x}_1\overline{x}_2x_4 \vee \overline{x}_1\overline{x}_3x_4 \vee \overline{x}_2\overline{x}_3x_4 \vee x_1x_2x_3 \vee x_1x_3\overline{x}_4 \vee x_2x_3\overline{x}_4$ has two equivalence symmetries, $E\{x_1, x_4\}$ and $E\{x_2, x_4\}$. The Karnaugh map for this function is shown in Figure 2.20.

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 0 | 1 | 1 |
| 01 | 0 | 1 | 0 | 1 |
| 11 | 0 | 1 | 1 | 0 |
| 10 | 0 | 1 | 0 | 1 |

**Figure 2.20.** *The Karnaugh map for the function* $f(x_4, x_3, x_2, x_1) = \overline{x}_1\overline{x}_2\overline{x}_3 \vee \overline{x}_1\overline{x}_2x_4 \vee \overline{x}_1\overline{x}_3x_4 \vee \overline{x}_2\overline{x}_3x_4 \vee x_1x_2x_3 \vee x_1x_3\overline{x}_4 \vee x_2x_3\overline{x}_4$

Figure 2.21 shows the Karnaugh maps for the 6 equivalence symmetries that may exist in a 4-variable Boolean function.

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | $a$ | $b$ | $c$ | $d$ |
| 01 | | | | |
| 11 | $a$ | $b$ | $c$ | $d$ |
| 10 | | | | |

$E\{x_1, x_2\}$

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | $a$ | | | $c$ |
| 01 | | $a$ | $c$ | |
| 11 | | $b$ | $d$ | |
| 10 | $d$ | | | $d$ |

$E\{x_1, x_3\}$

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | $a$ | $c$ | | |
| 01 | | | $c$ | $a$ |
| 11 | | | $d$ | $b$ |
| 10 | $b$ | $d$ | | |

$E\{x_1, x_4\}$

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | $a$ | | | $c$ |
| 01 | $b$ | | | $d$ |
| 11 | | $b$ | $d$ | |
| 10 | | $a$ | $c$ | |

$E\{x_2, x_3\}$

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | $a$ | $c$ | | |
| 01 | $b$ | $d$ | | |
| 11 | | | $d$ | $b$ |
| 10 | | | $c$ | $a$ |

$E\{x_2, x_4\}$

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | $a$ | | $a$ | |
| 01 | $b$ | | $b$ | |
| 11 | $c$ | | $c$ | |
| 10 | $d$ | | $d$ | |

$E\{x_3, x_4\}$

**Figure 2.21.** *The Karnaugh maps showing the necessary patterns for the 6 equivalence symmetries.*

### Nonequivalence Symmetries

Nonequivalence symmetries are similar to equivalence symmetries except that the values of $x_ix_j$ are changed from 01 to 10. That is, a function $f(x_n, \ldots, x_i, \ldots, x_j, \ldots, x_1)$ is said to have a nonequivalence symmetry in $\{x_i, x_j\}$ if

$$f(x_n, \ldots, 0, \ldots, 1, \ldots, x_1) = f(x_n, \ldots, 1, \ldots, 0, \ldots, x_1) \tag{2.12}$$

This is written $N\{x_i, x_j\}$. Figure 2.22 shows the 6 possible nonequivalence symmetries for a 4-variable Boolean function.

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | |
| 01 | $a$ | $b$ | $d$ | $c$ |
| 11 | | | | |
| 10 | $a$ | $b$ | $d$ | $c$ |

$N\{x_1, x_2\}$

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | $a$ | $c$ | |
| 01 | $a$ | | | $c$ |
| 11 | $b$ | | | $d$ |
| 10 | | $b$ | $d$ | |

$N\{x_1, x_3\}$

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | $c$ | $a$ |
| 01 | $a$ | $c$ | | |
| 11 | $b$ | $d$ | | |
| 10 | | | $d$ | $b$ |

$N\{x_1, x_4\}$

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | $a$ | $c$ | |
| 01 | | $b$ | $d$ | |
| 11 | $b$ | | | $d$ |
| 10 | $a$ | | | $c$ |

$N\{x_2, x_3\}$

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | $c$ | $a$ |
| 01 | | | $d$ | $b$ |
| 11 | $b$ | $d$ | | |
| 10 | $a$ | $c$ | | |

$N\{x_2, x_4\}$

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | $a$ | | $a$ |
| 01 | | $b$ | | $b$ |
| 11 | | $d$ | | $d$ |
| 10 | | $c$ | | $c$ |

$N\{x_3, x_4\}$

**Figure 2.22.** *The Karnaugh maps showing the necessary patterns for the 6 nonequivalence symmetries.*

### Single Variable Symmetries

A third type of degree two symmetry occurs when the function's values remain unchanged if $x_ix_j$ is changed from 10 to 11 or from 00 to 01. This first single variable symmetry is written $S\{x_j|x_i\}$ and is defined as

$$f(x_n, \ldots, 1, \ldots, 0, \ldots, x_1) = f(x_n, \ldots, 1, \ldots, 1, \ldots, x_1) \tag{2.13}$$

while the second single variable symmetry is written $S\{x_j|\overline{x}_i\}$ and is defined as

$$f(x_n, \ldots, 0, \ldots, 0, \ldots, x_1) = f(x_n, \ldots, 0, \ldots, 1, \ldots, x_1) \tag{2.14}$$

There are 24 possible single variable symmetries for a 4-variable function.

## 2.8 Conclusion

This chapter provides background concepts to aid in the understanding of the material presented in the rest of this dissertation. An overview of switching functions and logic synthesis is given in Sections 2.1 and 2.2, followed by details on various representations of switching functions in Section 2.3. Particular attention is paid to the spectral and autocorrelation representations in Sections 2.4 and 2.5, as all of the following chapters focus on these concepts. Section 2.6 introduces the concept of classifying switching functions, and explains the uses of different classification techniques. Chapter 6 expands on this area with a classification technique based on the autocorrelation coefficients of switching functions. The final section consists of an introduction to symmetries of switching functions, which is also expanded upon later in the dissertation (Chapter 4).

# Chapter 3

# Properties of the Autocorrelation Coefficients

## 3.1 Introduction

In this chapter we formulate and prove a number of theorems relating the autocorrelation coefficients to their underlying switching functions. The motivation behind this chapter is to identify patterns in the autocorrelation coefficients that help simplify the task of logic synthesis. For instance, being able to identify a degenerate function may save considerable processing time in the synthesis of a function. This will be of particular use when extending this work to multiple-output functions. Currently, a BDD representation of a single-output function automatically removes any redundant variables. However, in a shared BDD representation for a multiple-output function this will not necessarily occur and so identification of degnerate functions via another method is required. Similarly, minimization for sparse functions may be approached differently if this information is known before attempting logic synthesis. Another motivation is that knowledge of the properties inherent to the coefficients may allow us to determine a more efficient computational approach.

The material in this chapter begins with notation that is used through-out this and subsequent chapters. There next follows a section discussing the connections between the spectral and autocorrelation coefficients. Section 3.4 derives equations for converting between the autocorrelation coefficients computed using $\{0,1\}$ encoding for the function's outputs and those computed using $\{+1,-1\}$ encoding, and Section 3.5 examines various properties of the coefficients using both encodings. The chapter closes with a concluding section that discusses potential uses of the properties proven in this chapter.

## 3.2 Notation

The notation described here is used in the the following sections and in subsequent chapters.

- The term "a $\{0,1\}/\{+1,-1\}$ coefficient" refers to an autocorrelation coefficient computed using the given encoding for the function's outputs.
- $k$ is the number of true minterms in the function.
- The variable ordering $x_n, ..., x_1$ is used through-out. Thus a coefficient $B(001)$ is the first order coefficient corresponding to $x_1$.
- $\tau$ and $\tau'$ indicate values ranging through 0 to $2^n - 1$. $\tau_a$ is used to indicate one such value.
- $\tau_i$ refers to a value whose binary expansion contains a 1 in the $i^{th}$ bit, while the remaining $n-1$ bits are 0.
- $\tau_{i\alpha}$ refers to a set of values for which the binary expansion contains a 1 in the $i^{th}$ bit while the remaining $n-1$ bits have the value $\alpha \in \{0, ..., 2^{n-1}-1\}$. $\tau_{\bar{i}\alpha}$ refers to a set of values for which the binary expansion contains a 0 in the $i^{th}$ bit while the remaining $n-1$ bits have the value $\alpha$.
- $|\tau|$ is the weight, or the number of ones in the binary expansion of $\tau$. If $|\tau| = j$ then $B(\tau)$ and $C(\tau)$ are said to be $j^{th}$ order coefficients.

## 3.3 Relationship Between Spectral and Autocorrelation Coefficients

Karpovsky demonstrates in [26] the use of the Wiener-Khinchin theorem in the computation of the autocorrelation coefficients from the spectral coefficients. The equation for this computation is given as Equation 3.1.

$$B = \frac{1}{2^n} \times T^n \times R^2 \tag{3.1}$$

In this equation $R^2$ is the vector of spectral coefficients with each element squared. Note that $S$ may be used in place of $R$ in order to compute $C$. This relationship is used in Section 3.4 to derive equations for converting between $B$ and $C$.

It is interesting to note that the elements of $R$ ($S$) must be squared. It is for this reason that the autocorrelation transform is not reversible – some sign information is lost in the squaring process.

## 3.4 Converting Between $B$ and $C$

In Chapter 2.4 one reason for the use of $\{+1, -1\}$ encoding was discussed. In this section we derive equations for converting from one encoding to the other. In these equations the $\{0, 1\}$ encoded outputs of the function are labeled $z_i$ while the $\{+1, -1\}$ encoded outputs are labeled $y_i$, the $\{0, 1\}$ encoded spectral coefficients are labeled $r_u$ while the $\{+1, -1\}$ coefficients are labeled $s_u$, and the $\{0, 1\}$ encoded autocorrelation coefficients are labeled $B(\tau)$ while the $\{+1, -1\}$ coefficients are labeled $C(\tau)$.

The following equation describes the relationship between $z_i$ and $y_i$:

$$y_i = -2z_i + 1 \tag{3.2}$$

Based on Equation 3.2 and Equation 2.1 ($R = T^n \times Z$) we find that $s_i$ and $r_i$ are related as follows.

$$\begin{aligned} s_i &= -2r_i \\ s_0 &= -2r_0 + 2^n \end{aligned} \tag{3.3}$$

This relationship combined with Equation 3.1 leads to Equation 3.4:

$$C(\tau) = 2^n - 4k + 4B(\tau). \tag{3.4}$$

## 3.5 Properties

The first properties to be examined are those relating to the range of possible values that any autocorrelation coefficient may assume. These are identified and explained in Section 3.5.1. This section also discusses the value of the sum of the autocorrelation coefficients, as well as properties for the identification of the constant functions.

The next section introduces a number of theorems that relate patterns in the autocorrelation coefficients to sparse functions (or their inverse), as well as theorems that allow the identification of sparse functions from the examination of the autocorrelation coefficients. There is, however, a large group of functions for which no particular patterns exist in the autocorrelation coefficients. Section 3.5.3 makes some observations about this group of functions.

Section 3.5.4 introduces two theorems that relate the first and second order autocorrelation coefficients to the existence of XOR logic within the function. These properties have particular use in the areas of three-level minimization and decomposition, as well as in the design of KDDs. These applications are addressed in Chapter 7.

The final section on properties of the autocorrelation coefficients is Section 3.5.5. This section theorizes that coefficients of different orders may be related, and further, that the computation of coefficients of a lower order may allow for a reduction in the necessary operations when computing coefficients of a higher order. This theory is proven, and discussions on the applicability of this technique are presented.

### 3.5.1 General Observations on the Signs and Values of the Coefficients

There are a number of restrictions on the values of both the $\{0, 1\}$ and $\{+1, -1\}$ autocorrelation coefficients. Knowing these limitations provides for a simple check as to whether the coefficients have been computed correctly. They also lead to the identification of more specific properties relating the coefficient values to the original switching function.

**Lemma 3.1** $B(\tau) \in \{0, \ldots, 2^n\} \ \forall \ \tau$.

**Lemma 3.2** $B(\tau)$ *is even* $\forall \ \tau \neq 0$.

The above lemmas are clear from the definition of the autocorrelation function.

**Lemma 3.3** $B(\tau) \leq B(0) \ \forall \ \tau \neq 0$ *and* $B(0) = k$.

**Proof.**

Since

$$B(0) = \sum_{v=0}^{2^n-1} f(v) \times f(v \oplus 0)$$

and

$$f(v \oplus 0) = f(v) \ \forall \ v$$

then every true minterm results in a sum term of $1 \times 1$ and every false minterm results in a sum term of $0 \times 0$. Thus the the total value of the summation is the number of true minterms for the function. The only way another coefficient $B(\tau)$ could have the same value is if $f(v \oplus \tau) = f(v) \ \forall \ v$. ■

**Lemma 3.4** $C(\tau) \in \{-2^n, \ldots, 2^n\}$ *and is evenly divisible by 2* $\forall \ \tau$.

The largest possible value for $C(\tau)$ is obtained when $f(v) = f(v \oplus \tau) \ \forall \ v$, giving a result of $2^n$. The smallest possible value for $C(\tau)$ is obtained when $f(v) = -f(v \oplus \tau) \ \forall \ v$, giving a result of $-2^n$.

**Lemma 3.5** *A function may have at most* $2^{n-1}$ *negative* $C(\tau)$.

**Proof.**

Let us define a function $f(X)$ for which there are $2^{n-1}$ negative coefficients. Without loss of generality we assume that for this function every $C(\tau)$, $2^{n-1} \leq \tau \leq 2^n - 1$, is negative. If $2^{n-1} \leq \tau \leq 2^n - 1$ then in the autocorrelation equation

$$0 \leq v \leq 2^{n-1} - 1 \;\Rightarrow\; 2^{n-1} \leq v \oplus \tau \leq 2^n - 1$$

and

$$2^{n-1} \leq v \leq 2^n - 1 \;\Rightarrow\; 0 \leq v \oplus \tau \leq 2^{n-1} - 1.$$

In other words, in computing each of the negative coefficients we are matching a minterm from the top half of the function with one from the bottom half of the function.

For any one of the designated coefficients to be negative, there must be $2^{n-2} + 1$ of the values $0 \leq v \leq 2^{n-1} - 1$ negative if the values in $2^{n-1} \leq v \leq 2^n - 1$ are positive, or vice versa. However, this results in the remaining $2^{n-1}$ coefficients having positive values. Thus there can be at most $2^{n-1}$ negative autocorrelation coefficients. ■

**Lemma 3.6** $C(\tau) \leq C(0)\ \forall\ \tau \neq 0$ *and* $C(0) = 2^n$.

**Proof.**

This can be determined by applying Equation 3.4 to Lemma 3.3. We can also explain the result as follows. Since in $\{+1, -1\}$ encoding $f(v) \times f(v) = 1\ \forall\ v$ then

$$\begin{aligned} C(0) &= \sum_{v=0}^{2^n-1} f(v) \times f(v \oplus 0) \\ &= \sum_{v=0}^{2^n-1} 1 \\ &= 2^n. \end{aligned}$$

■

The next theorem defines the sum of the $2^n$ coefficients.

**Theorem 3.1**

$$\sum_{\tau=0}^{2^n-1} B(\tau) = k^2. \tag{3.5}$$

**Lemma 3.7**

$$\sum_{tau=1}^{2^n-1} B(\tau) = 2\binom{k}{2}. \tag{3.6}$$

$\binom{k}{2}$ is the number of pairings of the minterms as computed in the summation of the autocorrelation coefficients. This is then multiplied by 2 to produce all possible pairings in the form $i, j$ and $j, i$.

**Proof.**

Using Lemma 3.7 the sum of all of the $\{0, 1\}$ autocorrelation coefficients is as follows:

$$\begin{aligned}
\sum_{\tau=0}^{2^n-1} B(\tau) &= B(0) + 2\binom{k}{2} \\
&= k + 2\tfrac{k(k-1)}{2} \\
&= k^2.
\end{aligned}$$

■

**Corollary 3.1.1**

$$\sum_{\tau=1}^{2^n-1} C(\tau) = (2^n - 2k)^2. \tag{3.7}$$

**Proof.**

By substituting the expression from Theorem 3.1 into the equation for converting between $C(\tau)$ and $B(\tau)$ (Equation 3.4) we find the following:

$$\begin{aligned}\sum_{\tau=0}^{2^n-1} C(\tau) &= \sum_{\tau=0}^{2^n-1} (2^n - 4k + 4B(\tau)) \\ &= \sum_{\tau=0}^{2^n-1} 2^n - 4\sum_{\tau=0}^{2^n-1} k + 4\sum_{\tau=0}^{2^n-1} B(\tau) \\ &= (2^n)^2 - 2^n 4k + 4k^2 \\ &= (2^n - 2k)^2.\end{aligned}$$

■

The next theorem deals with the situation in which all the coefficients have the same value.

**Theorem 3.2** *$f(X) = 1$ or $f(X) = 0$ if and only if $C(\tau) = C(\tau')$ $\forall$ $\tau$ and $\tau' \in \{0, \ldots, 2^n - 1\}$.*

**Proof.**

If all the coefficients are equal, they must all have the value $2^n$ as the coefficient $C(0)$ always has this value. Based on this, if all of the coefficients have equal value, then this implies that the function matches itself at every value of $\tau$. This can only occur if the function consists entirely of true minterms, or entirely of false minterms.

■

**Corollary 3.2.1** *$f(X) = 0$ or $f(X) = 1$ if and only if $B(\tau) = B(\tau')$ $\forall$ $\tau$ and $\tau' \in \{0, \ldots, 2^n - 1\}$.*

**Proof.**

By applying the equation for converting between $C(\tau)$ and $B(\tau)$ (Equation 3.4) to Theorem 3.2 we find that there are two solutions:

$$B(\tau) = 0 \text{ or}$$
$$B(\tau) = k.$$

If $B(\tau) = 0 \; \forall \; \tau$ then $B(0) = 0$ and the function is then $f(X) = 0$. If $f(X) = 0$ then clearly $B(\tau) = 0 \; \forall \; \tau$.

If $B(\tau) = k \; \forall \; \tau$ then if $k = 2^n$ then the function is $f(X) = 1$. If $k < 2^n$ then there must exist a coefficient such that $B(\tau) < k$, since there will be some coefficient where one of the sum terms is of the form $f(v) \times f(v \oplus \tau)$ where $f(v) = 1$ while $f(v \oplus \tau) = 0$. Then not all of the coefficients are equal. So to have $B(\tau) = k \; \forall \; \tau$, $k$ must be $2^n$. It is clear from the autocorrelation equation that if $f(X) = 1$ then $B(\tau) = 2^n \; \forall \; \tau$.

■

### 3.5.2 Theorems for Small Numbers of Dissimilar Minterms

Theorems 3.3, 3.4 and 3.5 are the first to be discussed in this section. These theorems pertain to the situation that occurs when there are a majority of true minterms and few false minterms for the function, or the inverse. Theorems 3.3 and 3.4 refer to two special cases, while Theorem 3.5 gives results for the general case.

Additional theorems in this section discuss the properties of the autocorrelation coefficients when exactly half of the minterms are true (Theorem 3.6). Since one of the possibilities for this situation is for the function to be degenerate, an additional theorem, Theorem 3.7, generalizes on how to identify degenerate functions from their autocorrelation coefficents.

**Theorem 3.3** *A function f(X) has exactly one dissimilar minterm if and only if* $C(\tau) = 2^n - 4 \; \forall \; \tau \neq 0$.

**Proof.**

In $\{+1, -1\}$ encoding there is no distinction between exactly one true minterm and the inverse situation. This is because negation of the function has no effect on the coefficients' values when using $\{+1, -1\}$ encoding.

Without loss of generality let us define a function $f$ such that

$$f(v) = 1,\ v \in\ 0, \ldots, 2^n - 2$$
$$f(v) = -1,\ v = 2^n - 1.$$

Then

$$\begin{aligned}
C(\tau) &= \sum_{v=0}^{2^n-1} f(v) \times f(v \oplus \tau) \\
&= (\sum_{v=0}^{2^n-2} f(v) \times f(v \oplus \tau)) + f(2^n - 1) \times f(2^n - 1 \oplus \tau) \\
&= (\sum_{v=0}^{2^n-2} 1 \times f(v \oplus \tau)) + -1 \times 1 \\
&= (2^n - 2 - 1) - 1 \\
&= 2^n - 4\ \forall\ \tau \neq 0.
\end{aligned}$$

Thus if $f(X)$ has exactly one true minterm then all of the coefficients $C(\tau) = 2^n - 4$, $\tau \neq 0$.

For the second part of this proof, if all that is known of the function is the coefficients of this pattern, then it can be shown as follows that the function must have either exactly one true or exactly one false minterm.

For a coefficient $C(\tau)$ let us define $q$ as the number of positive pairs in the summation, and $r$ as the number of negative pairs in the summation. A pair in this case is a combination of two minterms $i, j$, and a positive pair results when both minterms are true or when both are false. It should be noted that in the summation for the autocorrelation equation each pair is encountered twice. Then

$$2q - 2r = 2^n - 4$$

and

$$2q + 2r = 2^n.$$

These equations can be solved to show that $r = 1$. If there is only one negative pair in the summation then there is only one pair combining a true and a false minterm; all other pairs must combine either two true minterms or two false minterms. If there is only one coefficient $C(\tau)$ for which this holds, then there can be any number of combinations of true and false minterms to meet these requirements. However, there are $2^n - 1$ coefficients that have only one negative pair; therefore there can be only one dissimilar minterm in the function. ■

**Corollary 3.3.1** *A function $f(X)$ has exactly one dissimilar minterm if and only if $B(\tau) = k - 1$.*

**Proof.** The above is determined by substituting $C(\tau) = 2^n - 4$ into the conversion equation $C(\tau) = 2^n - 4k + 4B(\tau)$. ■

### Explanations

Although Corollary 3.3.1 states a general result in terms of $k$, in practice the values for $B(\tau)$ are quite limited. This is because for a function to have exactly one dissimilar minterm then either $k = 2^n - 1$, in which case $B(\tau) = 2^n - 2 \; \forall \; \tau \neq 0$, or $k = 1$, which results in $B(\tau) = 0 \; \forall \; \tau \neq 0$. Figure 3.1 illustrates this situation.

Theorem 3.3 deals only with the situation where the function has exactly one dissimilar minterm. Theorem 3.4 goes on to extend this a little further.

**Theorem 3.4** *A function has exactly two dissimilar minterms if and only if*

$$\begin{aligned} &C(0) = 2^n, \\ &C(\tau_a) = 2^n, \text{ and} \\ &C(\tau) = 2n - 8 \; \forall \; \tau \textit{ and } \tau_a \neq 0 \textit{ and } \tau \neq \tau_a. \end{aligned}$$

| | functions | | autocorrelation coefficients | |
|---|---|---|---|---|
| | (i) | (ii) | (i) | (ii) |
| 000 | 0 | 1 | 1 | 7 |
| 001 | 0 | 1 | 0 | 6 |
| 010 | 0 | 1 | 0 | 6 |
| 011 | 0 | 1 | 0 | 6 |
| 100 | 0 | 1 | 0 | 6 |
| 101 | 0 | 1 | 0 | 6 |
| 110 | 1 | 0 | 0 | 6 |
| 111 | 0 | 1 | 0 | 6 |

**Figure 3.1.** *Two three-variable functions demonstrating the situation for which (i) $B(0) = 1$ and (ii) $B(0) = 2^n - 1$.*

**Proof.**

We approach this proof by first demonstrating that if there is one coefficient $C(\tau_a) = 2^n, \tau \neq 0$ and the remaining $2^n - 2$ coefficients $C(\tau) = 2^n - 8$, then the function has exactly two dissimilar minterms. Let us define a function $f$ such that

$$f(v) = 1,\ v \in 0, \dots, 2^n - 1,\ v \neq i, j$$
$$f(v) = -1,\ v = i, j.$$

Without loss of generality let $i = 0$ and $j = 1$. Then

$$\begin{aligned} C(\tau) &= \sum_{v=0}^{2^n-1} f(v) \times f(v \oplus \tau) \\ &= f(i) \times f(i \oplus \tau) + f(j) \times f(j \oplus \tau) + \sum_{v=2}^{2^n-1} f(v) \times f(v \oplus \tau) \\ &= -1 \times f(0 \oplus \tau) + -1 \times f(1 \oplus \tau) + \sum_{v=2}^{2^n-1} 1 \times f(v \oplus \tau) \end{aligned}$$

Then if $i \oplus \tau = j$ and $j \oplus \tau = i$, $C(\tau) = 2^n$. Otherwise $C(\tau) = -2 + (2^n - 4) - 2 = 2^n - 8$. Because of the nature of the $\oplus$ operator, $i \oplus \tau = j \ \Leftrightarrow \ j \oplus \tau = i$, and so there is only one assignment of $\tau$ for which this can occur.

A similar process to that shown in the proof of Theorem 3.3 can be used to prove that this pattern of coefficients can only result in a function with exactly two dissimilar minterms. ■

**Corollary 3.4.1** *A function has exactly two dissimilar minterms if and only if*

$$B(0) = B(\tau_a) = k \text{ and}$$
$$B(\tau) = k - 2 \;\forall\; \tau \textit{ and } \tau_a \neq 0 \textit{ and } \tau \neq \tau_a.$$

**Proof.**

The above is determined by substituting the results of Theorem 3.4 into the conversion equation $C(\tau) = 2^n - 4k + 4B(\tau)$. ■

**Explanation**

Once again, although Corollary 3.4.1 states a general result, in practice the values are limited to the following:

(i) $B(0) = B(\tau_a) = 2$ and
$B(\tau) = 0$, or

(ii) $B(0) = B(\tau_a) = 2^n - 2$ and
$B(\tau) = 2^n - 4$.

It should also be noted that this pattern of coefficients indicates that the function is either itself degenerate or is related through the application of the autocorrelation invariance operators to a degenerate function. The autocorrelation invariance operators are discussed in detail in Chapter 6.

Finally, given the location of one of the true minterms, the information contained in the non-zero autocorrelation coefficient allows us to identify the second of the true minterms. This can be determined by finding the value of the inputs at the known true minterm's location, and XOR-ing this value with the value of $\tau$ for the non-zero coefficient.

The results in Theorems 3.3 and 3.4 may be generalized as follows:

**Theorem 3.5** *A function has $d$ dissimilar minterms if and only if the autocorrelation coefficients have the following properties:*

- *$C(0) = 2^n$,*
- *for $\binom{d}{p}$ $p \in 2, 4, 6, \ldots, d$, (or $2, 4, 6, \ldots, d-1$ if $d$ is odd) $C(\tau) = 2^n - 4d + 4p$, and*
- *for the remaining coefficients, $C(\tau) = 2^n - 4d$.*

**Proof.**

The proof is similar to those for Theorems 3.3 and 3.4. Let us define a function $f(X)$ for which there are $d$ dissimilar minterms. Without loss of generality we assume that

$$f(v) = -1, v \in 0, \ldots, d-1 \text{ and}$$
$$f(v) = 1, v \in d, \ldots, 2^n - 1.$$

Then there are $d-p$ mod 2 ways (resulting in $\binom{d}{2}+\binom{d}{4}+\ldots+\binom{d}{d-1}$ or $\binom{d}{d}$ coefficients) in which pairs of dissimilar minterms may match up, resulting in

$$\begin{aligned} C(\tau) &= 2p - 2(d-p) + \sum_{d}^{2^n-1-d} f(v) \times f(v \oplus \tau) \\ &= 2p - 2(d-p) + 2^n - 2d \\ &= 4p - 4d + 2^n \end{aligned}$$

where the first term $2p$ is the result of the sum of the matching dissimilar minterms, the second term $2(d-p)$ is the result of the sum of the non-matching dissimilar minterms, and the final term is the sum of the remaining minterms which are all similar. There are also coefficients resulting from the situation in which none of the dissimilar coefficients match in the summation:

$$\begin{aligned} C(\tau) &= -2d + \sum_{d}^{2^n-1-d} f(v) \times f(v \oplus \tau) \\ &= 2^n - 4d. \end{aligned}$$

Again, using a similar technique to that shown in the proof of Theorem 3.3, if $q$ is the number of positive pairs and $r$ is the number of negative pairs then

$$2q + 2r = 2^n \text{ and}$$
$$2q - 2r = 2^n - 4d$$

which results in $r = d$.

■

The next situation to take into account occurs when the number of true minterms and the number of false minterms are equal.

**Theorem 3.6** *A function $f(X)$ has $2^{n-1}$ autocorrelation coefficients $C(\tau) = 2^n$ (including $C(0)$) and the remaining $2^{n-1}$ coefficients $C(\tau') = -2^n$ if and only if the function has exactly $2^{n-1}$ true minterms.*

**Proof.**

We first demonstrate that if a function is either itself dependent on only one of its input variables (or related through the application of the autocorrelation invariance operators to a function that is dependent on only one of its input variables) then half of its coefficients have the value $2^n$ while the other half have the value $-2^n$. A function that is dependent on only one input variable must have half of the minterms true and half of them false. Without loss of generality let us define $f(X) = x_1$ where $x_1$ is the lowest order bit of the input $X$. Then if $\tau$ is an odd number the binary expansion of $\tau$ contains a 1 in the lowest order bit, and then by definition $f(v) = \overline{f(v \oplus \tau)}$. Then

$$\begin{aligned} C(\tau) &= \sum_{v=0}^{2^n-1} 1 \times -1 \\ &= -2^n. \end{aligned}$$

Similarly if $\tau'$ is an even number, then the binary expansion contains a 0 in the lowest order bit and by definition $f(v) = f(v \oplus \tau')$. Then

$$\begin{aligned} C(\tau') &= \sum_{v=0}^{2^n-1} (-)1 \times (-)1 \\ &= 2^n. \end{aligned}$$

Next we demonstrate that given autocorrelation coefficients of the pattern described above the function must be dependent on only one of the input variables (or related to such a function). Without loss of generality let us assume that $C(\tau') = 2^n$ where $\tau'$ is even and $C(\tau) = -2^n$ where $\tau$ is odd. $C(\tau') = 2^n$ where $\tau'$ is even indicates that the function matches up two false or two true minterms for every product in the summation. Additionally every product being computed is comparing two inputs for which $x_1$ remains unchanged. Moreover, $C(\tau) = -2^n$ where $\tau$ is odd indicates that the function matches a false minterm with a true minterm for every product in the summation, and that every product is matching a pair of inputs for which $x_1$ varies. Based on this we can determine that the function must be dependent only on $x_1$, and so there must be $2^{n-1}$ true minterms in the function. ■

**Corollary 3.6.1** *A function $f(X)$ has $2^{n-1}$ autocorrelation coefficients $B(\tau) = 2^{n-1}$ (including $B(0)$) and the remaining $2^{n-1}$ coefficients $B(\tau') = 0$ if and only if the function has exactly $2^{n-1}$ true minterms.*

**Proof.**

This can be determined from applying the conversion equation to the results of Theorem 3.6. This is also trivially clear since $B(0) = k$, and $B(0) = 2^{n-1}$. ■

Theorem 3.6 identifies one situation where the function may be degenerate. The next theorem, Theorem 3.7 extends this to the general case.

**Theorem 3.7** *A function $f(X)$ is independent of $j$ of its input variables if and only if $C(\tau_i) = 2^n$ $\forall$ $i$ $\in 1..n$ such that the function does not depend on variable $x_i$.*

**Proof.**

Without loss of generality let us define a function $f(X)$ that is independent of $x_n$. By definition, $f(0, x_{n-1}, \ldots, x_1) = f(1, x_{n-1}, \ldots, x_1)$. Then

$$\begin{aligned} C(\tau_n) &= \sum_{v=0}^{2^n-1} f(v) \times f(v \oplus \tau_n) \\ &= \sum_{v=0}^{2^{n-1}-1} f(v) \times f(v \oplus \tau_n) + \sum_{v=2^{n-1}}^{2^n-1} f(v) \times f(v \oplus \tau_n). \end{aligned}$$

Let us define the range 0 *to* $2^{n-1} - 1$ as A and $2^{n-1}$ *to* $2^n - 1$ as B. Then

$$v \in A \;\Rightarrow\; v \oplus \tau_n \in B \text{ and}$$

$$v \in B \;\Rightarrow\; v \oplus \tau_n \in A.$$

Since the function is defined to have $f(A) = f(B)$ then

$$\begin{aligned} C(\tau_n) &= \sum_{v=0}^{2^{n-1}-1} f(v) \times f(v \oplus \tau_n) + \sum_{v=2^{n-1}}^{2^n-1} f(v) \times f(v \oplus \tau_n) \\ &= \sum_{v=0}^{2^{n-1}-1} 1 + \sum_{v=2^{n-1}}^{2^n-1} 1 \\ &= 2^n. \end{aligned}$$

To prove the second part of the theorem we define (without loss of generality) a function $f(X)$ for which $C(\tau_n) = 2^n$. This is only possible if $f(v) = f(v \oplus \tau_n)\ \forall\ v$. This implies that $f(1, x_{n-1}, \ldots, x_1) = f(0, x_{n-1}, \ldots, x_1)$, indicating that $f(X)$ is not dependent on $x_n$.

■

**Corollary 3.7.1** *A function $f(X)$ is independent of $j$ of its input variables if and only if $B(\tau_i) = k\ \forall\ i \in 1..n$ such that the function does not depend on variable $x_i$.*

**Proof.**

The results from Theorem 3.7 may be extended to $\{0,1\}$ encoding through the application of Equation 3.4, $C(\tau) = 2^n - 4k + 4B(\tau)$.

■

### 3.5.3 Observations About Functions For Which $2 < B(0) < 2^n - 2$

For functions with equal or almost equal numbers of true and false minterms it is difficult to predict the pattern of the higher order coefficients. However, some observations may be made:

- Functions with $B(0) = q$ belong to a class of functions that is related by output negation to another class of functions with $B(0) = 2^n - q$. However, as the number of variables for the functions increases, there will be more than one class with the same value for $B(0)$. Identification of the related classes will in such a case require examination of the higher order coefficients as well as $B(0)$. This cannot be extended to the $\{+1,-1\}$ encoded coefficients, for two reasons. The first is that $C(0)$ always has the value $2^n$, therefore examination of this coefficient does not provide any information except the number of variables in the function. The second reason is that the $\{+1,-1\}$ encoded autocorrelation coefficients do not change when the function is negated, and in fact functions that fall into different $\{0,1\}$ autocorrelation classes may be combined into one larger $\{+1,-1\}$ class.
- Certain classes are also related to each other through the application of the type (v) spectral invariance operation. This is discussed further in Chapter 6.

### 3.5.4 Identification of Exclusive-OR Logic

Although it is difficult to give patterns for the entire spectrum of coefficients for functions with equal or nearly equal numbers of true and false minterms, it is possible to identify some information about the type of logic within the function through the identification of the first and second order coefficients. The theorems in this section expand on this idea.

**Theorem 3.8** $C(\tau_i) = -2^n$ *if and only if the function* $f(X)$ *has a decomposition*

$$f(X) = f^*(X) \oplus x_i$$

*where* $f^*(X)$ *is independent of* $x_i$.

**Proof.**

We first determine that a function with the decomposition $f(X) = f^*(X) \oplus x_i$ has a first order autocorrelation coefficient $C(\tau_i) = -2^n$. Without loss of generality let $i = n$. Then

$$\begin{aligned}
C(\tau_n) &= \sum_{v=0}^{2^n-1} f(v) \times f(v \oplus \tau_n) \\
&= \sum_{v=0}^{2^n-1} ([f^*(v) \oplus x_n] \times [f^*(v \oplus \tau_n)) \oplus (x_n \oplus \tau_n)]) \\
&= \sum_{v=0}^{2^{n-1}-1} (f^*(v) \oplus 0) \times (f^*(v \oplus \tau_n) \oplus (0 \oplus \tau_n)) + \\
&\quad \sum_{v=2^{n-1}}^{2^n-1} (f^*(v) \oplus 1) \times (f^*(v \oplus \tau_n) \oplus (1 \oplus \tau_n)) \\
&= \sum_{v=0}^{2^{n-1}-1} (f^*(v) \oplus 0) \times (f^*(v \oplus \tau_n) \oplus (1)) + \\
&\quad \sum_{v=2^{n-1}}^{2^n-1} (-f^*(v)) \times (f^*(v \oplus \tau_n) \oplus (0))
\end{aligned}$$

$$= \sum_{v=0}^{2^{n-1}-1} f^*(v) \times (-f^*(v \oplus \tau_n)) +$$

$$\sum_{v=2^{n-1}}^{2^n-1} (-f^*(v)) \times f^*(v \oplus \tau_n)$$

$$= -\sum_{v=0}^{2^n-1} f^*(v) \times f^*(v \oplus \tau_n)$$

$$= -2^n$$

since by definition $f^*(X)$ is independent of $x_n$.

We next determine that a first order $\{+1, -1\}$ autocorrelation coefficient with the value $-2^n$ implies that the function $f(X)$ can be decomposed into $f^*(X) \oplus x_i$. If $C(\tau_i) = -2^n$ then in the equation

$$C(\tau_i) = \sum_{v=0}^{2^n-1} f(v) \times f(v \oplus \tau_i)$$

$f(v) = -f(v \oplus \tau_i)\ \forall\ v$ implying that $f(v) = -f(v \oplus \tau_i)\ \forall\ v$. This means that half of the function is the inverse of the other half, which can be achieved by defining a function $f(X)$ as

$$f(X) = f^*(X) \oplus x_i.$$

■

If no first order coefficients meet the requirements for the presence of this decomposition, we then go on to include the second order coefficients in the examination. This is described in Theorem 3.9.

**Theorem 3.9** $C(\tau_i) = C(\tau_j) = C(\tau_{ij}) = 0,\ i \neq j$ *if and only if the function* $f(X)$ *can be decomposed into* $f^*(X) \oplus g(X)$ *where* $g(X) = x_i * x_j,\ * \in \{\wedge, \vee\}$ *and* $f^*(X)$ *is independent of both* $x_i$ *and* $x_j$.

**Proof.**

Let us define a function $f(X) = f^*(X) \oplus g(X)$ where $g(X) = x_i \wedge x_j$ and $f^*(X)$ is independent of $x_i$ and $x_j$, and let us assume without loss of generality that $i = n$ and $j = n - 1$. Then

$$\begin{aligned} C(\tau) &= \sum_{v=0}^{2^n-1} f(v) \times f(v \oplus \tau) \\ &= \sum_A f(v_1) \times f(v_1 \oplus \tau) + \\ &\quad \sum_B f(v_2) \times f(v_2 \oplus \tau) + \\ &\quad \sum_C f(v_3) \times f(v_3 \oplus \tau) + \\ &\quad \sum_D f(v_4) \times f(v_4 \oplus \tau) \end{aligned}$$

where

A: $v_1 = 0$ to $2^{n-2} - 1$ $(0000 \ldots 0011)$,

B: $v_2 = 2^{n-2}$ to $2^{n-1} - 1$ $(0100 \ldots 0111)$,

C: $v_3 = 2^{n-1}$ to $2^n - 2^{n-2} - 1$ $(1000 \ldots 1011)$, and

D: $v_4 = 2^n - 2^{n-2}$ to $2^n - 1$ $(1100 \ldots 1111)$[1].

[1]Four variable expansions are given for the sake of clarity only. This does not limit the proof to four variables.

Then

$$\begin{aligned} C(\tau_{n-1}) &= \sum_A f(v_1) \times f(v_1 \oplus \tau_{n-1}) + \\ &\quad \sum_B f(v_2) \times f(v_2 \oplus \tau_{n-1}) + \\ &\quad \sum_C f(v_3) \times -f(v_3 \oplus \tau_{n-1}) + \\ &\quad \sum_D - f(v_4) \times f(v_4 \oplus \tau_{n-1}) \\ &= 2^{n-2} + 2^{n-2} + -2^{n-2} + -2^{n-2} \\ &= 0 \end{aligned}$$

and similarly for $C(\tau_n)$ and $C(\tau_{n\ n-1})$.

If $C(\tau_n) = C(\tau_{n-1}) = C(\tau_{n\ n-1}) = 0$ then each of the summations may be broken down into

$$C(\tau) = 2^{n-2} + 2^{n-2} - 2^{n-2} - 2^{n-2}.$$

Let us assume there exists some variable ordering such that

$$\begin{aligned} &\sum_A f(v_1) \times f(v_1 \oplus \tau_{n-1}) = 2^n - 2 \text{ and} \\ &\sum_B f(v_2) \times f(v_2 \oplus \tau_{n-1}) = 2^n - 2 \text{ and} \\ &\sum_C f(v_3) \times f(v_3 \oplus \tau_{n-1}) = -2^n - 2 \text{ and} \\ &\sum_D f(v_4) \times f(v_4 \oplus \tau_{n-1}) = -2^n - 2. \end{aligned}$$

Then the first two summations tell us that for part of the function $f(v)$ is independent of variable $x_{n-1}$ and the second two indicate that for part of the function $f(v)$ contains $\oplus\ x_{n-1}$. This indicates that the solution must be of the form

$$f(X) = f^*(X) \oplus g(X)$$

where $f^*(X)$ is independent of $x_{n-1}$ and $g(X)$ contains $x_{n-1}$. The same process is then applied to the other known coefficients, $C(\tau_{n\ n-1}) = C(\tau_n) = 0$. There are two possible solutions:

**Solution 1**

$$\begin{array}{lrrr}
 & \tau_{n-1} & \tau_n & \tau_{n\ n-1} \\
\sum_A f(v_1) \times f(v_1 \oplus \tau) & = 2^n & = 2^n & = -2^n \\
\sum_B f(v_2) \times f(v_2 \oplus \tau) & = 2^n & = -2^n & = 2^n \\
\sum_C f(v_3) \times f(v_3 \oplus \tau) & = -2^n & = 2^n & = 2^n \\
\sum_D f(v_4) \times f(v_4 \oplus \tau) & = -2^n & = -2^n & = -2^n
\end{array}$$

The above is obtained for $g(X) = x_1 \wedge x_2$.

**Solution 2**

$$\begin{array}{lrrr}
 & \tau_{n-1} & \tau_n & \tau_{n\ n-1} \\
\sum_A f(v_1) \times f(v_1 \oplus \tau) & = -2^n & = -2^n & = -2^n \\
\sum_B f(v_2) \times f(v_2 \oplus \tau) & = -2^n & = 2^n & = 2^n \\
\sum_C f(v_3) \times f(v_3 \oplus \tau) & = 2^n & = -2^n & = 2^n \\
\sum_D f(v_4) \times f(v_4 \oplus \tau) & = 2^n & = 2^n & = 2^n
\end{array}$$

The above is obtained for $g(X) = x_1 \vee x_2$.

■

Theorems 3.8 and 3.9 are put to various uses in Chapter 7.

### 3.5.5 Relating Coefficients of Different Orders

This section makes an observation about the autocorrelation coefficients that has the potential for use in their computation. Although it is not possible to directly compute one autocorrelation coefficient from the final values of any other coefficients, it is possible to make use of intermediate sum terms from the computation other coefficients in computing additional coefficients.

In $\{+1,-1\}$ encoding there exist the properties

$$f(x) = f(y) \Leftrightarrow f(x) \times f(y) = 1$$

and

$$f(x) \neq f(y) \Leftrightarrow f(x) \times f(y) = -1.$$

Therefore if we have three values $f(x), f(y)$ and $f(z)$ then

$$\begin{aligned} f(x) \times f(y) \times f(x) \times f(z) &= f(x) \times f(x) \times f(y) \times f(z) \\ &= 1 \times f(y) \times f(z) \\ &= f(y) \times f(z). \end{aligned}$$

The autocorrelation function may be expanded as follows:

$$\begin{aligned} C(\tau) &= \sum_{v=0}^{2^n-1} f(v) \times f(v \oplus \tau) \\ &= [f(0) \times f(0 \oplus \tau)] + [f(1) \times f(1 \oplus \tau)] + \cdots + [f(2^n-1) \times f(2^n-1 \oplus \tau)]. \end{aligned}$$

If the individual product terms of the summations for various values of $\tau$ are stored, they then can be combined (multiplied) to produce products belonging to summations for other values of $\tau$.

For example, given a three-variable function $f(X)$ as defined in Table 3.1, the summations for the three first-order coefficients for $f(X)$ are

$$\begin{aligned} C(001) &= ab + ba + cd + dc + ef + fe + gh + hg \\ &= 2(ab + cd + ef + gh) \\ C(010) &= 2(ac + bd + eg + fh) \\ C(100) &= 2(ae + bf + cg + dh). \end{aligned}$$

| $x_3x_2x_1$ | $f(X)$ |
|---|---|
| 000 | $a$ |
| 001 | $b$ |
| 010 | $c$ |
| 011 | $d$ |
| 100 | $e$ |
| 101 | $f$ |
| 110 | $g$ |
| 111 | $h$ |

**Table 3.1.** *A generic three-variable function $f(X)$ with unknown outputs.*

Examination of the products in the summation for $C(011)$

$$C(011) \quad = 2(ad + bc + eh + fh)$$

shows that this can be rewritten as a combination of products from $C(001)$ and $C(010)$ using the properties defined at the start of this section:

$$\begin{aligned} C(011) \quad &= 2(ad + bc + eh + fh) \\ &= 2(ab(ac + bd) + ef(eg + fh)) \\ \text{OR} \quad &= 2(ac(ab + cd) + eg(ef + gh)) \end{aligned}$$

or other combinations of the product terms from $C(001)$ and $C(010)$. Similarly, given that $C(111) = 2(ah + bg + cf + de)$, $C(011)$ can be computed as $C(011) = 2(ah(ae+dh)+cf(bf+cg))$ or other combinations of terms from $C(100)$ and $C(111)$.

Based on this example there is a clear connection between the different coefficient values. The relationship can be generalized as follows in Theorem 3.10.

**Theorem 3.10** *If $\tau_z = \tau_x \oplus \tau_y$ then $C(\tau_z)$ can be computed from the intermediate products of $C(\tau_x)$ and $C(\tau_y)$.*

**Proof.** The proof of this theorem also makes use of various properties of the XOR operator:

$$\begin{aligned} \text{if} \quad & a = b \oplus c \text{ then} \\ & c = a \oplus b \text{ and} \\ & b = a \oplus c. \end{aligned}$$

Additionally,

$$a \oplus a = b.$$

Given

$$C(\tau_x) = \sum_{v_x=0}^{2^n-1} f(v_x) \times f(v_x \oplus \tau_x)$$

$$C(\tau_y) = \sum_{v_y=0}^{2^n-1} f(v_y) \times f(v_y \oplus \tau_y)$$

$$C(\tau_z) = \sum_{v_z=0}^{2^n-1} f(v_z) \times f(v_z \oplus \tau_z)$$

and

$$\tau_z = \tau_x \oplus \tau_y$$

then for the above relationship to be true, for every assignment to $v_z$ there must exist an expression

$$f(v_y) \times f(v_y \oplus \tau_y) \oplus f(v_x) \times f(v_x \oplus \tau_x)$$

such that

1. two of the terms are equal and thus cancel, and
2. of the other two terms,

   one of the terms is equal to $f(v_z)$ and

   the other term is equal to $f(v_z \oplus z)$.

Since $a = b \Rightarrow f(a) = f(b)$ we reduce this to demonstrating that $\exists\ v_y \times (v_y \oplus \tau_y) \oplus v_x \times (v_x \oplus \tau_x)$ with the required properties.

There are two basic options: either $v_x = v_y$ or $v_x \neq v_y$.

1. If $v_y = v_x$ then

$$\begin{aligned} &v_y \oplus \tau_y = v_z \text{ and} \\ &v_x \oplus \tau_x = v_z \oplus \tau_z. \end{aligned}$$

   These can be combined as follows

$$\begin{aligned} &v_x \oplus \tau_x = v_y \oplus \tau_y \oplus \tau_z \\ &v_x \oplus v_y \oplus \tau_x = \tau_y \oplus \tau_z \\ &\tau_x = \tau_y \oplus \tau_z \\ &\tau_z = \tau_y \oplus \tau_x \end{aligned}$$

2. If $v_x = v_y$ then either $v_y \oplus \tau_y = v_x$ or $v_x \oplus \tau_x = v_y$.

   (a) $v_y \oplus \tau_y = v_x$ means that either $v_y = v_z$ or $v_y = v_z \oplus z$.

      i. $v_y = v_z$ means that $v_x \oplus \tau_x = v_z \oplus z$. Combine the two to get

$$\begin{aligned} &v_x \oplus \tau_x = v_y \oplus \tau_z \\ &v_x \oplus v_y = \tau_x \oplus \tau_z \\ &\text{using } v_y \oplus \tau_y = v_x \text{ gives} \\ &\tau_y = \tau_x \oplus \tau_z \end{aligned}$$

      ii. $v_y = v_z \oplus \tau_z$ means that $v_x \oplus \tau_x = v_z$. Combine the two to get

$$\begin{aligned} &v_y = v_x \oplus \tau_x \oplus \tau_z \\ &v_y \oplus v_x = \tau_x \oplus \tau_z \\ &\tau_y = \tau_x \oplus \tau_z \end{aligned}$$

   (b) $v_x \oplus \tau_x = v_y$ means that either $v_x = v_z$ and $v_y \oplus \tau_y = v_z \oplus \tau_z$ or $v_x = v_z \oplus \tau_z$ and $v_y \oplus \tau_y = v_z$. Arguments similar to those above show that both situations result in $\tau_x \oplus \tau_y = \tau_z$.

■

Not only is it possible to reuse the individual products, but it is also possible to use certain sums of the product terms in subsequent coefficient computations. This is apparent in the first example where

$$\begin{aligned}
C(001) &= 2(ab + cd + ef + gh) \\
&= 2((ab + cd) + (ef + gh)) \\
C(010) &= 2(ac + bd + eg + fh) \\
C(011) &= 2(ad + bc + eh + fg) \\
&= 2(ac(ab + cd) + eg(ef + gh))
\end{aligned}$$

$(ab + cd)$ and $(ef + gh)$ have already been computed and the final result of each sum can be used in the computation of $C(011)$. There is the potential to reduce the number of computations required to to compute $C(011)$ from $1+2^{n-1}$ multiplications and $2^{n-1}-1$ additions to $1+2^{n-2}$ multiplications and $2^{n-2}-1$ additions, a reduction by half.

However, there are a number of problems with incorporating this technique into a computation method. The first problem is that once some grouping of a coefficient's intermediate products has been used in a subsequent coefficient's computation, that grouping will never be of use for any other computation. Since there are $n$ first-order coefficients and $\binom{n}{2}$ second-order coefficients, for $n > 3$ there will be more second-order coefficients than first-order, and thus there are not enough groupings to reuse. Following this pattern, it would be necessary to perform all $2^n$ computations for the third-order coefficients in order to achieve savings of approximately half $\binom{n}{3}$ of the fourth-order coefficients, and so on. Thus the computations for some of the higher-order coefficients could be reduced by half, but at best every odd-order would require the full number of computations.

The second problem is in the overhead required. Storage for the intermediate product terms and for pair-wise sums are required for all $n$ first-order coefficients while computation of the first and second-order coefficients are performed. A determination of which pairs to combine must be made, and then storage must be reclaimed for the next set of computations. There will be a small amount of memory overhead, but more importantly, a large amount of computation overhead. As is shown in

Chapter 5 the use of decision diagrams as data structures can generate the coefficients so quickly that any technique not making use of this approach, even with some savings in computation, cannot compete.

The third problem is that all $2^n$ outputs of the function are required. Most practical computation techniques use only the on-set of the function, in most cases greatly reducing the number of values that must be taken into account. It would be very difficult for this technique to incorporate this.

## 3.6 Conclusion

This chapter derives a number of new results demonstrating that the autocorrelation coefficients have some very useful properties that can be used in the identification of various types of switching functions. The types of functions identified include sparse functions and degenerate functions including the constant functions. Knowledge that a function is sparse or degenerate is very useful in logic synthesis, as it allows the designer to approach the synthesis process in a manner tailored to the type of function.

The chapter also proves theorems relating the first and second order autocorrelation coefficients to the presence of XOR logic within a function. Given the recent interest in three-level function representations and extensions to BDDs such as KDDs, determining the existence of XOR logic within a function clearly has many applications. These applications are discussed further in Chapter 7, with some implementations based on these properties that show very promising results.

Finally, the investigations in this chapter were partially motivated by the need for an efficient computation technique. This chapter investigates whether or not computation of a coefficient could be performed by reusing either all or part of the sum comprising the value of a previously computed technique.

# Chapter 4

# Symmetries and Autocorrelation Coefficients

## 4.1 Introduction

Symmetries have been used in many types of logic synthesis applications, as referenced in Chapter 2. For these uses, it is desirable to identify the existence of a particular symmetry before beginning the synthesis process. Section 4.5 explains how it is possible to determine a test for the existence of many symmetries, based on subsets of the spectral coefficients.

Much of the work in this chapter grew from an investigation into whether the autocorrelation coefficients could be used to develop a similar test for the existence of symmetries. This was hypothesized based on the fact that the two types of coefficients are closely related; in fact, Equation 3.1 demonstrates how the autocorrelation coefficients may be computed from the spectral coefficients. Although we cannot determine a *test* for the symmetries based on the autocorrelation coefficients, certain patterns in the autocorrelation coefficients are a required *condition* for the existence of symmetries. That is, the autocorrelation coefficients will always take on certain patterns if the symmetries exist within the function, but properties other than the existence of symmetries may also result in the same autocorrelation patterns. The investigations into this are detailed in Sections 4.2, 4.3 and 4.4, and further explanations into why we cannot test for symmetries using the autocorrelation coefficients are given in Section 4.5.

Also during these investigations a new type of symmetry was discovered. We label these new symmetries *antisymmetries*. One of the popular uses of symmetries as cited in Chapter 2 is in the reduction of decision diagrams, and it is clear that the antisymmetries have great potential in this area. Having discovered these symmetries, we derive spectral conditions and tests for them and detail the resulting autocorrelation coefficient properties.

## 4.2 Totally Symmetric Functions

A totally symmetric function is a function for which the output is unchanged for *any* permutation of the input variables. Majority functions fall into this class of function, and many totally symmetric functions may be realized using a single *vertex* or *majority* gate [11]. Additionally, totally symmetric functions can be represented using ROBDDs that are at worst quadratic and at best linear in the number of inputs [3]. Figure 4.1 shows an example of a Boolean function that is totally symmetric.

| $x_3x_4 \backslash x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | $a$ | $b$ | $c$ | $b$ |
| 01 | $b$ | $c$ | $d$ | $c$ |
| 11 | $c$ | $d$ | $e$ | $d$ |
| 10 | $b$ | $c$ | $d$ | $c$ |

**Figure 4.1.** *The Karnaugh map for a totally symmetric 4-variable Boolean function.*

**Theorem 4.1** *If a function $f(X)$ is totally symmetric then the autocorrelation coefficients $B(\tau) = B(\tau')\ \forall\ \tau$ and $\tau'$ such that $|\tau| = |\tau'|$*[1].

[1] All theorems and proofs in this chapter make use of the notation defined in Section 3.2.

**Proof.**

Let us define a totally symmetric function $f(X)$. Then by definition $\forall\ v_a, v_b \in \{0, ..., 2^n - 1\}$, $f(v_a) = f(v_b)$ if and only if $|v_a| = |v_b|$. Then

$$\begin{aligned}
B(\tau) &= \sum_{v=0}^{2^n-1} f(v) \times f(v \oplus \tau) \\
&= \sum_{|v|=0} f(v) \times f(v \oplus \tau) \ + \sum_{|v|=1} f(v) \times f(v \oplus \tau) \ + \cdots + \sum_{|v|=n} f(v) \times f(v \oplus \tau) \\
&= f(0) \times f(\tau) + \sum_{r=1}^{n-1} [f(|v| = r) \sum_{|v|=r} f(v \oplus \tau)] + f(2^n - 1) \times f(2^n - 1 \oplus \tau)
\end{aligned}$$

and similarly for $B(\tau')$. By definition, $f(\tau) = f(\tau')$ and so

$$f(0) \times f(\tau) = f(0) \times f(\tau') \qquad \text{and}$$
$$f(2^n - 1) \times f(2^n - 1 \oplus \tau) = f(2^n - 1) \times f(2^n - 1 \oplus \tau').$$

In comparing individual sum terms for the remaining part of the summation it is clear that $f(v \oplus \tau)$ may not always be equal to $f(v \oplus \tau')$. For example, let $v = 1$, $\tau = 1$ and $\tau' = 2$, then $v \oplus \tau = 0$ and $v \oplus \tau' = 3$. However, within the sum over $|v| = 1$ $v = 2$ will also be assigned. Since within a summation over $|v| = r$ for some $r$ $v$ takes on all possible values with weight $r$, then for some value $v_i$ in the summation there is always another value $v_j$ for which $v_i \oplus \tau = v_j \oplus \tau'$. Then

$$\sum_{|v|=r} f(v \oplus \tau) = \sum_{|v|=r} f(v \oplus \tau')$$

and so $B(\tau) = B(\tau')$.

■

## 4.3 (Non)Equivalence Symmetries

As indicated in Chapter 2, totally symmetric functions are extremely rare. We therefore examine a less restrictive form of symmetries, symmetries of degree 2. Of the symmetries of degree 2 we first examine the equivalence and nonequivalence symmetries.

A function
$f(x_n, ..., x_i, ..., x_j, ..., x_1)$ is said to possess an equivalence symmetry $E\{x_i, x_j\}$ if

$$f(x_n, ..., 0, ..., 0, ..., x_1) = f(x_n, ..., 1, ..., 1, ..., x_1).$$

and is said to possess a nonequivalence symmetry $N\{x_i, x_j\}$ if

$$f(x_n, ..., 0, ..., 1, ..., x_1) = f(x_n, ..., 1, ..., 0, ..., x_1).$$

The easiest way to illustrate such symmetries is through the use of Karnaugh maps. For example, for the equivalence symmetry $E\{x_1, x_2\}$, we can draw a Karnaugh map for any function with this symmetry as follows:

| $x_3x_4 \backslash x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | **a** | $e$ | **a** | $i$ |
| 01 | **b** | $f$ | **b** | $j$ |
| 11 | **c** | $g$ | **c** | $k$ |
| 10 | **d** | $h$ | **d** | $l$ |

The nonequivalence symmetry $N\{x_1, x_2\}$ has a similar Karnaugh map:

| $x_3x_4 \backslash x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | $e$ | **a** | $i$ | **a** |
| 01 | $f$ | **b** | $j$ | **b** |
| 11 | $g$ | **c** | $k$ | **c** |
| 10 | $h$ | **d** | $l$ | **d** |

Based on these examples we hypothesize that the existence of (non)equivalence symmetries are reflected in a function's autocorrelation coefficients.

**Theorem 4.2** *$B(\tau_{\,i\bar{j}\alpha}) = B(\tau_{\,\bar{i}j\alpha})$, $i, j \in \{1, ..., n\}$ and $i \neq j$ if a function $f(X)$ possesses an equivalence symmetry $E\{x_i, x_j\}$ or a nonequivalence symmetry $N\{x_i, x_j\}$.*

**Proof.**

Without loss of generality let us define a function $f(X)$ such that $f(X)$ possesses the equivalence symmetry $E\{x_n, x_{n-1}\}$. Then if

$$\begin{aligned}
A &\in 0 \; to \; 2^{n-2} - 1 \; (0000...0011) \\
B &\in 2^{n-2} \; to \; 2^{n-1} - 1 \; (0100...0111) \\
C &\in 2^{n-1} \; to \; 2^{n} - 2^{n-2} - 1 \; (1000...1011) \\
D &\in 2^{n} - 2^{n-2} \; to \; 2^{n} - 1 \; (1100...1111)
\end{aligned}$$
[2]

the resulting computation for the autocorrelation coefficients is

$$\begin{aligned}
B(\tau_{n\;\overline{n-1}\;\alpha}) &= \sum_{v=0}^{2^n-1} f(v) \times f(v \oplus \tau_{n\;\overline{n-1}\;\alpha}) \\
&= \sum_{v\;\in\;A} f(v) \times f(v \oplus \tau_{n\;\overline{n-1}\;\alpha}) \;+\; \sum_{v\;\in\;B} f(v) \times f(v \oplus \tau_{n\;\overline{n-1}\;\alpha}) \\
&\quad + \sum_{v\;\in\;C} f(v) \times f(v \oplus \tau_{n\;\overline{n-1}\;\alpha}) \;+\; \sum_{v\;\in\;D} f(v) \times f(v \oplus \tau_{n\;\overline{n-1}\;\alpha})
\end{aligned}$$

and similarly for $B(\tau_{\overline{n}\;n-1\;\alpha})$.

$v \in A \Rightarrow v \oplus \tau_{n\;\overline{n-1}\;\alpha} \in C$,

$v \in B \Rightarrow v \oplus \tau_{n\;\overline{n-1}\;\alpha} \in D$,

$v \in C \Rightarrow v \oplus \tau_{n\;\overline{n-1}\;\alpha} \in A$, and

$v \in D \Rightarrow v \oplus \tau_{n\;\overline{n-1}\;\alpha} \in B$

while

$v \in A \Rightarrow v \oplus \tau_{\overline{n}\;n-1\;\alpha} \in B$,

$v \in B \Rightarrow v \oplus \tau_{\overline{n}\;n-1\;\alpha} \in A$,

$v \in C \Rightarrow v \oplus \tau_{\overline{n}\;n-1\;\alpha} \in D$, and

$v \in D \Rightarrow v \oplus \tau_{\overline{n}\;n-1\;\alpha} \in C$.

For the remainder of this proof we denote $f(v)$ for $v \in A$ as $f(A)$, and similarly for $B, C$, and $D$. Then

$$\begin{aligned} B(\tau_{n\ \overline{n-1}\ \alpha}) &= \sum_{v\ \in\ A} f(A) \times f(C) + \sum_{v\ \in\ B} f(B) \times f(D) \\ &\quad + \sum_{v\ \in\ C} f(C) \times f(A) + \sum_{v\ \in\ D} f(D) \times f(B) \\ &= 2\sum_{v\ \in\ A} f(A) \times f(C) + 2\sum_{v\ \in\ B} f(B) \times f(D) \end{aligned}$$

and

$$\begin{aligned} B(\tau_{\overline{n}\ n-1\ \alpha}) &= \sum_{v\ \in\ A} f(A) \times f(B) + \sum_{v\ \in\ B} f(B) \times f(A) \\ &\quad + \sum_{v\ \in\ C} f(C) \times f(D) + \sum_{v\ \in\ D} f(D) \times f(C) \\ &= 2\sum_{v\ \in\ A} f(A) \times f(B) + 2\sum_{v\ \in\ B} f(C) \times f(D). \end{aligned}$$

By definition, $f(A) = f(D)$, and so

$$\begin{aligned} B(\tau_{n\ \overline{n-1}\ \alpha}) &= 2\sum_{v\ \in\ A} f(D) \times f(C) + 2\sum_{v\ \in\ B} f(B) \times f(A) \\ &= B(\tau_{\overline{n}\ n-1\ \alpha}). \end{aligned}$$

If $f(X)$ contains a nonequivalence symmetry $N\{x_n, x_{n-1}\}$ then $f(B) = f(C)$ and so

$$\begin{aligned} B(\tau_{n\ \overline{n-1}\ \alpha}) &= 2\sum_{v\ \in\ A} f(A) \times f(B) + 2\sum_{v\ \in\ B} f(C) \times f(D) \\ &= B(\tau_{\overline{n}\ n-1\ \alpha}). \end{aligned}$$

■

**Corollary 4.2.1** *$C(\tau_{\ i\bar{j}\alpha}) = C(\tau_{\ \bar{i}j\alpha})$, $i, j\ \in\ \{1, ..., n\}$ if the function $f(X)$ possesses an equivalence symmetry $E\{x_i, x_j\}$ or a nonequivalence symmetry $N\{x_i, x_j\}$.*

**Proof.** The encoding of $f(X)$ does not affect the above proof. ∎

## 4.4 Single Variable Symmetries

The autocorrelation coefficients of Boolean functions with single variable symmetries also have identifiable properties. A function is said to possess a single variable symmetry $S\{x_j|x_i\}$ if

$$f(x_n, ..., 1, ..., 0, ..., x_1) = f(x_n, ..., 1, ..., 1, ..., x_1)$$

and is said to possess a single variable symmetry $S\{x_j|\overline{x}_i\}$ if

$$f(x_n, ..., 0, ..., 0, ..., x_1) = f(x_n, ..., 0, ..., 1, ..., x_1).$$

Figure 4.2 shows the Karnaugh maps for two Boolean functions, each demonstrating one type of single variable symmetry.

| $x_3x_4 \backslash x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | $e$ | $i$ | **a** | **a** |
| 01 | $f$ | $j$ | **b** | **b** |
| 11 | $g$ | $k$ | **c** | **c** |
| 10 | $h$ | $l$ | **d** | **d** |

$S\{x_2|x_1\}$

$f(1, 0, x_3, x_4) = f(1, 1, x_3, x_4)$

| $x_3x_4 \backslash x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | **a** | **a** | $e$ | $i$ |
| 01 | **b** | **b** | $f$ | $j$ |
| 11 | **c** | **c** | $g$ | $k$ |
| 10 | **d** | **d** | $h$ | $l$ |

$S\{x_2|\overline{x}_1\}$

$f(0, 0, x_3, x_4) = f(0, 1, x_3, x_4)$

**Figure 4.2.** *The definitions and Karnaugh maps of two types of single variable symmetries for 4-variable Boolean functions.*

**Theorem 4.3** *$B(\tau_{i\bar{j}\alpha}) = B(\tau_{ij\alpha})$, $i, j \in \{1..n\}$ and $i \neq j$ if the function $f(X)$ possesses a single variable symmetry $S\{x_j|x_i\}$ or $S\{x_j|\overline{x}_i\}$.*

**Proof.**

Without loss of generality let us define a function $f(X)$ such that $f(X)$ possesses the single variable symmetry $S\{x_{n-1}|x_n\}$. We again use the ranges $A, B, C$ and $D$ and notation $f(A), f(B), f(C)$ and $f(D)$ as defined in the previous section.

$$\begin{aligned}
v \ &\in \ A \Rightarrow v \oplus \tau_{n\ \overline{n-1}\ \alpha} \ \in \ C, \\
v \ &\in \ B \Rightarrow v \oplus \tau_{n\ \overline{n-1}\ \alpha} \ \in \ D, \\
v \ &\in \ C \Rightarrow v \oplus \tau_{n\ \overline{n-1}\ \alpha} \ \in \ A, \text{ and} \\
v \ &\in \ D \Rightarrow v \oplus \tau_{n\ \overline{n-1}\ \alpha} \ \in \ B
\end{aligned}$$

while

$$\begin{aligned}
v \ &\in \ A \Rightarrow v \oplus \tau_{n\ n-1\ \alpha} \ \in \ D, \\
v \ &\in \ B \Rightarrow v \oplus \tau_{n\ n-1\ \alpha} \ \in \ C, \\
v \ &\in \ C \Rightarrow v \oplus \tau_{n\ n-1\ \alpha} \ \in \ B, \text{ and} \\
v \ &\in \ D \Rightarrow v \oplus \tau_{n\ n-1\ \alpha} \ \in \ A,
\end{aligned}$$

and so

$$\begin{aligned}
B(\tau_{n\ \overline{n-1}\ \alpha}) \ &= \ \sum_{v\ \in\ A} f(A) \times f(C) + \sum_{v\ \in\ B} f(B) \times f(D) \\
&\quad + \sum_{v\ \in\ C} f(C) \times f(A) + \sum_{v\ \in\ D} f(D) \times f(B) \\
&= \ 2\sum_{v\ \in\ A} f(A) \times f(C) + 2\sum_{v\ \in\ B} f(B) \times f(D)
\end{aligned}$$

and

$$\begin{aligned}
B(\tau_{n\ n-1\ \alpha}) \ &= \ \sum_{v\ \in\ A} f(A) \times f(D) + \sum_{v\ \in\ B} f(B) \times f(C) \\
&\quad + \sum_{v\ \in\ C} f(C) \times f(B) + \sum_{v\ \in\ D} f(D) \times f(A) \\
&= \ 2\sum_{v\ \in\ A} f(A) \times f(D) + 2\sum_{v\ \in\ B} f(B) \times f(C).
\end{aligned}$$

By definition, $f(C) = f(D)$ and so

$$\begin{aligned} B(\tau_{n\ \overline{n-1}\ \alpha}) &= 2\sum_{v\ \in\ A} f(A) \times f(D) + 2\sum_{v\ \in\ B} f(B) \times f(C) \\ &= B(\tau_{n\ n-1\ \alpha}). \end{aligned}$$

If $f(X)$ possesses $S\{x_{n-1}|\overline{x}_n\}$ then $f(A) = f(B)$ and so

$$\begin{aligned} B(\tau_{n\ \overline{n-1}\ \alpha}) &= 2\sum_{v\ \in\ A} f(B) \times f(C) + 2\sum_{v\ \in\ B} f(A) \times f(D) \\ &= B(\tau_{n\ n-1\ \alpha}). \end{aligned}$$

■

Again, the proof is equivalent for $\{+1, -1\}$ encoding.

## 4.5 Testing for Symmetries

According to [5] there exist a set of tests that can be performed on subsets of a function's spectral coefficients to determine the existence of a particular type of symmetry. These tests are described in Table 4.1 using the following notation:

$S^0$ includes all spectral coefficients that involve neither of $x_i$ or $x_j$,

$S^1$ includes all spectral coefficients that involve $x_i$ but not $x_j$,

$S^2$ includes all spectral coefficients that involve $x_j$ but not $x_i$, and

$S^3$ includes all spectral coefficents that involve both $x_i$ and $x_j$.

The theorems in Sections 4.2, 4.3 and 4.4 are the results of attempting to determine similar tests for symmetries based on subsets of the autocorrelation coefficients. As these demonstrate, the autocorrelation coefficients provide necessary conditions for the existence of the various symmetries. However, it can be shown that these conditions on the autocorrelation coefficients are not sufficient for the existence of the symmetries. A 3-variable example having the required pattern of $B(\tau_{\ i\bar{j}\alpha}) = B(\tau_{\ \bar{i}j\alpha})$

| Symmetry | Test |
|---|---|
| $S\{x_i\|\overline{x}_j\}$ | $S^1 + S^3 = 0$ |
| $S\{x_j\|\overline{x}_i\}$ | $S^2 + S^3 = 0$ |
| $E\{x_i, x_j\}$ | $S^1 + S^2 = 0$ |
| $N\{x_i, x_j\}$ | $S^1 - S^2 = 0$ |
| $S\{x_j\|x_i\}$ | $S^2 - S^3 = 0$ |
| $S\{x_i\|x_j\}$ | $S^1 - S^3 = 0$ |

**Table 4.1.** *Spectral symmetry tests for symmetries in* $\{x_i, x_j\}$, $i < j$.

| $x_1 \backslash x_3x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 |

**Figure 4.3.** *An example of a 3-variable function that has* $B(\tau_{2\overline{3}\alpha}) = B(\tau_{\overline{2}3\alpha})$ *but does not contain either* $N\{x_2, x_3\}$ *or* $E\{x_2, x_3\}$

but not possessing either $E\{x_i, x_j\}$ or $N\{x_i, x_j\}$ is shown in Figure 4.3. This is due to the loss of sign information that is present in the spectral coefficients, but is lost in the autocorrelation coefficients.

## 4.6 Antisymmetries

### Definition

In this section we introduce six new types of symmetries called *antisymmetries*. Antisymmetries are thus named because instead of identifying two equal parts of the function's Karnaugh map they identify two parts of the Karnaugh map that are the exact inverse. With the help of inverters, this has great potential in the application of decision diagram reduction.

| Notation | Definition |
|---|---|
| $\overline{E}\{x_i, x_j\}$ | $f(x_n, ..., 0, 0, ..., x_1) = \overline{f(x_n, ..., 1, 1, ...x_1)}$ |
| $\overline{N}\{x_i, x_j\}$ | $f(x_n, ..., 0, 1, ..., x_1) = \overline{f(x_n, ..., 1, 0, ...x_1)}$ |
| $\overline{S}\{x_i \vert x_j\}$ | $f(x_n, ..., 1, 1, ..., x_1) = \overline{f(x_n, ..., 0, 1, ...x_1)}$ |
| $\overline{S}\{x_j \vert x_i\}$ | $f(x_n, ..., 1, 1, ..., x_1) = \overline{f(x_n, ..., 1, 0, ...x_1)}$ |
| $\overline{S}\{x_i \vert \overline{x}_j\}$ | $f(x_n, ..., 1, 0, ..., x_1) = \overline{f(x_n, ..., 0, 0, ...x_1)}$ |
| $\overline{S}\{x_j \vert \overline{x}_i\}$ | $f(x_n, ..., 0, 1, ..., x_1) = \overline{f(x_n, ..., 0, 0, ...x_1)}$ |

**Table 4.2.** *Definitions and notation for the antisymmetries of degree two.*

## Spectral Conditions on Functions possessing Antisymmetries

Like the equivalence, non-equivalence, and single-variable symmetries, there are tests that may be performed on the function's spectral coefficients that determine whether or not the symmetry in question is present. In this section we derive similar conditions and tests for the antisymmetries.

### Anti-equivalence Symmetries

**Definition 4.1** *A function $f(X)$ is said to possess an anti-equivalence symmetry $\overline{E}\{x_i, x_j\}$ if and only if*

$$f(x_n, ..., 0, ..., 0, ..., x_1) = \overline{f(x_n, ..., 1, ..., 1, ..., x_1)}.$$

### Derivation of Spectral Tests and Conditions

For this and subsequent derivations we use the following notation:

$f(0, 0, x_{n-2}, ..., x_1)$ is denoted $f_0$,

$f(0, 1, x_{n-2}, ..., x_1)$ is denoted $f_1$,

$f(1, 0, x_{n-2}, ..., x_1)$ is denoted $f_2$ and

$f(1, 1, x_{n-2}, ..., x_1)$ is denoted $f_3$.

Additionally, a similar notation is used when referring to the spectral coefficients for these functions:

The spectral coefficient vector for $f_0$ is denoted $S_0$,

the spectral coefficient vector for $f_1$ is denoted $S_1$,

the spectral coefficient vector for $f_2$ is denoted $S_2$ and

the spectral coefficient vector for $f_3$ is denoted $S_3$.

Without loss of generality we choose $i = n$ and $j = n - 1$. The anti-equivalence symmetry may be expressed using the above notation as $f_0 = \overline{f_3}$, which in turn may be expressed in terms of the spectral coefficients of $f_0$ and $\overline{f}_3$ as $S_0 = -S_3$. We can use the spectral vector of the function in this way because there is no loss of information in the transformation between the outputs of a Boolean function and its spectral coefficients, and we know that negating a function has the effect of negating all of its spectral coefficients [5].

[5] also shows that the subsets $S^0, S^1, S^2$ and $S^3$ are related to $S_0, S_1, S_2$, and $S_3$ in the following way:

$$\begin{aligned} 4S_0 &= S^0 + S^1 + S^2 + S^3 \\ 4S_1 &= S^0 - S^1 + S^2 - S^3 \\ 4S_2 &= S^0 + S^1 - S^2 - S^3 \\ 4S_3 &= S^0 - S^1 - S^2 + S^3 \end{aligned} \tag{4.1}$$

From this we can determine that the condition for the antisymmetry $\overline{E}\{x_{n-1}, x_n\}$ to exist is $S_0 = -S_3$ and the test for its existence is $S^0 = -S^3$.

### Anti-nonequivalence Symmetries

**Definition 4.2** *A function $f(X)$ is said to possess an anti-nonequivalence symmetry $\overline{N}\{x_i, x_j\}$ if and only if*

$$f(x_n, ..., 0, ..., 1, ..., x_1) = \overline{f(x_n, ..., 1, ..., 0, ..., x_1)}.$$

### Derivation of Spectral Tests and Conditions

Again, without loss of generality we choose $x_i = n$ and $x_j = n - 1$. Then we have $f_1 = \overline{f_2}$ and so $S_1 = -S_2$. From this we can determine that the condition for the antisymmetry $\overline{N}\{x_{n-1}, x_n\}$ to exist is $S_1 = -S_2$ and the test for its existence is $S^0 = S^3$.

### Anti-single variable Symmetries in $x_j$ over $x_i$

**Definition 4.3** *A function $f(X)$ is said to possess an anti-single variable symmetry $\overline{S}\{x_j|x_i\}$, $i < j$ if and only if*

$$f(x_n, ..., 0, ..., 1, ..., x_1) = \overline{f(x_n, ..., 1, ..., 1, ..., x_1)}.$$

### Derivation of Spectral Tests and Conditions

The following is the derivation of both the condition and test for the antisymmetry's existence, assuming a choice of $x_j = n$ and $x_i = n - 1$:

$$\begin{aligned} &f_1 = \overline{f_3} \\ &\Rightarrow S_1 = -S_3 \\ &\Rightarrow S^0 = S^1 \end{aligned}$$

### Anti-single variable Symmetry in $x_j$ over $\overline{x}_i$

**Definition 4.4** *A function $f(X)$ is said to possess an anti-single variable symmetry $\overline{S}\{x_j|\overline{x}_i\}$, $i < j$ if and only if*

$$f(x_n, ..., 0, ..., 0, ..., x_1) = \overline{f(x_n, ..., 1, ..., 0, ..., x_1)}.$$

### Derivation of Spectral Tests and Conditions

The following is the derivation of both the condition and test for the antisymmetry's existence, assuming a choice of $x_j = n$ and $x_i = n - 1$:

$$\begin{aligned} & f_0 = \overline{f_2} \\ & \Rightarrow S_0 = -S_2 \\ & \Rightarrow S^0 = -S^1 \end{aligned}$$

### Anti-single variable Symmetry in $x_i$ over $x_j$

**Definition 4.5** *A function $f(X)$ is said to possess an anti-single variable symmetry $\overline{S}\{x_i|x_j\}$, $i < j$ if and only if*

$$f(x_n, ..., 1, ..., 0, ..., x_1) = \overline{f(x_n, ..., 1, ..., 1, ..., x_1)}.$$

### Derivation of Spectral Tests and Conditions

The following is the derivation of both the condition and test for the antisymmetry's existence, assuming a choice of $x_j = n - 1$ and $x_i = n$:

$$\begin{aligned} & f_2 = \overline{f_3} \\ & \Rightarrow S_2 = -S_3 \\ & \Rightarrow S^0 = -S^2 \end{aligned}$$

### Anti-single variable Symmetry in $x_i$ over $\overline{x}_j$

**Definition 4.6** *A function $f(X)$ is said to possess an anti-single variable symmetry $\overline{S}\{x_i|\overline{x}_j\}$, $i < j$ if and only if*

$$f(x_n, ..., 0, ..., 0, ..., x_1) = \overline{f(x_n, ..., 0, ..., 1, ..., x_1)}.$$

### Deriviation of Spectral Tests and Conditions

The following is the derivation of both the condition and test for the antisymmetry's existence, assuming a choice of $x_j = n-1$ and $x_i = n$:

$$
\begin{aligned}
& f_0 = \overline{f_1} \\
& \Rightarrow S_0 = -S_1 \\
& \Rightarrow S^0 = -S^2
\end{aligned}
$$

### Summary of Conditions and Tests for the Antisymmetries of Degree Two

Table 4.3 summarizes the results derived above.

| Symmetry | Condition | Test |
|---|---|---|
| $\overline{E}\{x_i, x_j\}$ | $S_0 = -S_3$ | $S^0 = -S^3$ |
| $\overline{N}\{x_i, x_j\}$ | $S_0 = S_3$ | $S^0 = S^3$ |
| $\overline{S}\{x_i \vert x_j\}$ | $S_1 = -S_3$ | $S^0 = S^1$ |
| $\overline{S}\{x_j \vert x_i\}$ | $S_2 = -S_3$ | $S^0 = S^2$ |
| $\overline{S}\{x_i \vert \overline{x}_j\}$ | $S_0 = -S_2$ | $S^0 = -S^1$ |
| $\overline{S}\{x_j \vert \overline{x}_i\}$ | $S_0 = -S_1$ | $S^0 = -S^2$ |

**Table 4.3.** *Spectral conditions and tests for the antisymmetries of degree two.*

## Properties of Autocorrelation Coefficients for Functions possessing Antisymmetries

As is the case for the regular symmetries of degree 2, the existence of an antisymmetry imposes certain restrictions on the values of the autocorrelation coefficients. These restrictions are initially defined using $\{+1, -1\}$ encoding of the function, as they involve negation of the corresponding coefficients.

**Theorem 4.4** *$C(\tau_{i\alpha}) = -C(\tau_{j\alpha})$, $i, j \in \{1, ..., n\}$ and $i \neq j$ if the function $f(X)$ possesses an anti-equivalence symmetry $\overline{E}\{x_i, x_j\}$ or an anti-nonequivalence symmetry $\overline{N}\{x_i, x_j\}$.*

**Proof.**

Without loss of generality let us define a function $f(X)$ such that $f(X)$ possesses the anti-equivalence symmetry $\overline{E}\{x_n, x_{n-1}\}$. We will again use the ranges $A, ..., D$ and the corresponding notation $f(A), ..., f(D)$ as defined in Section 4.3. As before,

$v \in A \Rightarrow v \oplus \tau_n \in C$,

$v \in B \Rightarrow v \oplus \tau_n \in D$,

$v \in C \Rightarrow v \oplus \tau_n \in A$, and

$v \in D \Rightarrow v \oplus \tau_n \in B$

while

$v \in A \Rightarrow v \oplus \tau_{n-1} \in B$,

$v \in B \Rightarrow v \oplus \tau_{n-1} \in A$,

$v \in C \Rightarrow v \oplus \tau_{n-1} \in D$, and

$v \in D \Rightarrow v \oplus \tau_{n-1} \in C$, and then

$$\begin{aligned} C(\tau_{n\ \alpha}) &= \sum_{v \in A} f(A) \times f(C) + \sum_{v \in B} f(B) \times f(D) \\ &\quad + \sum_{v \in C} f(C) \times f(A) + \sum_{v \in D} f(D) \times f(B) \\ &= 2 \sum_{v \in A} f(A) \times f(C) + 2 \sum_{v \in B} f(B) \times f(D) \end{aligned}$$

while

$$\begin{aligned} C(\tau_{n-1\ \alpha}) &= \sum_{v\ \in\ A} f(A) \times f(B) + \sum_{v\ \in\ B} f(B) \times f(A) \\ &\quad + \sum_{v\ \in\ C} f(C) \times f(D) + \sum_{v\ \in\ D} f(D) \times f(C) \\ &= 2\sum_{v\ \in\ A} f(A) \times f(B) + 2\sum_{v\ \in\ B} f(C) \times f(D). \end{aligned}$$

By definition, $f(A) = -f(D)$, so

$$\begin{aligned} C(\tau_{n\ \alpha}) &= -2\sum_{v\ \in\ A} f(D) \times f(C) - 2\sum_{v\ \in\ B} f(B) \times f(A) \\ &= -C(\tau_{n-1\ \alpha}). \end{aligned}$$

If $f(X)$ contains an anti-nonequivalence symmetry $\overline{N}\{x_n, x_{n-1}\}$ then $f(B) = -f(C)$ and so

$$\begin{aligned} C(\tau_{n\ \alpha}) &= -2\sum_{v\ \in\ A} f(A) \times f(B) - 2\sum_{v\ \in\ B} f(C) \times f(D) \\ &= -C(\tau_{n-1\ \alpha}). \end{aligned}$$

■

**Corollary 4.4.1** *If $C(\tau) = -C(\tau')$ then*

$$B(\tau) = 2k - 2^{n-1} - B(\tau').$$

This is determined by using Equation 3.4, $C(\tau) = 2^n - 4k + 4B(\tau)$.

**Theorem 4.5** $C(\tau_{\ i\bar{j}\alpha}) = -C(\tau_{\ ij\alpha})$, $i, j\ \in\ \{1, ..., n\}$ *and* $i \neq j$ *if the function* $f(X)$ *possesses an anti-single variable symmetry* $\overline{S}\{x_j|x_i\}$ *or* $\overline{S}\{x_j|\overline{x}_i\}$.

**Proof.**

Without loss of generality let us define a function $f(X)$ such that $f(X)$ possesses the anti-single variable symmetry $\overline{S}\{x_{n-1}|x_n\}$. Then,

$$v \in A \Rightarrow v \oplus \tau_{n\ \overline{n-1}\ \alpha} \in C,$$

$$v \in B \Rightarrow v \oplus \tau_{n\ \overline{n-1}\ \alpha} \in D,$$

$$v \in C \Rightarrow v \oplus \tau_{n\ \overline{n-1}\ \alpha} \in A, \text{ and}$$

$$v \in D \Rightarrow v \oplus \tau_{n\ \overline{n-1}\ \alpha} \in B$$

while

$$v \in A \Rightarrow v \oplus \tau_{n\ n-1\ \alpha} \in D,$$

$$v \in B \Rightarrow v \oplus \tau_{n\ n-1\ \alpha} \in C,$$

$$v \in C \Rightarrow v \oplus \tau_{n\ n-1\ \alpha} \in B, \text{ and}$$

$$v \in D \Rightarrow v \oplus \tau_{n\ n-1\ \alpha} \in A, \text{ and then}$$

$$\begin{aligned}
C(\tau_{n\ \overline{n-1}\ \alpha}) &= \sum_{v\ \in\ A} f(A) \times f(C) + \sum_{v\ \in\ B} f(B) \times f(D) \\
&\quad + \sum_{v\ \in\ C} f(C) \times f(A) + \sum_{v\ \in\ D} f(D) \times f(B) \\
&= 2\sum_{v\ \in\ A} f(A) \times f(C) + 2\sum_{v\ \in\ B} f(B) \times f(D)
\end{aligned}$$

while

$$\begin{aligned}
C(\tau_{n\ n-1\ \alpha}) &= \sum_{v\ \in\ A} f(A) \times f(D) + \sum_{v\ \in\ B} f(B) \times f(C) \\
&\quad + \sum_{v\ \in\ C} f(C) \times f(B) + \sum_{v\ \in\ D} f(D) \times f(A) \\
&= 2\sum_{v\ \in\ A} f(A) \times f(D) + 2\sum_{v\ \in\ B} f(B) \times f(C).
\end{aligned}$$

By definition, $f(C) = -f(D)$, so

$$\begin{aligned}
C(\tau_{n\ \overline{n-1}\ \alpha}) &= -2\sum_{v\ \in\ A} f(A) \times f(D) - 2\sum_{v\ \in\ B} f(B) \times f(C) \\
&= -C(\tau_{n\ n-1\ \alpha}).
\end{aligned}$$

If $f(X)$ contains the anti-single variable symmetry $\overline{S}\{x_{n-1}|\overline{x}_n\}$ then $f(A) = -f(C)$ and so

$$\begin{aligned} C(\tau_{n\ \overline{n-1}\ \alpha}) &= -2\sum_{v\ \in\ A} f(A) \times f(B) - 2\sum_{v\ \in\ B} f(C) \times f(D) \\ &= -C(\tau_{n\ n-1\ \alpha}). \end{aligned}$$

■

**Corollary 4.5.1** *If* $C(\tau_{n\ \overline{n-1}\ \alpha}) = -C(\tau_{n\ n-1\ \alpha})$ *then*

$$B(\tau_{n\ \overline{n-1}\ \alpha}) = 2k - 2^{n-1} - B(\tau_{n\ n-1\ \alpha}).$$

This is determined by using Equation 3.4, $C(\tau) = 2^n - 4k + 4B(\tau)$.

## 4.7 Applications

The use of symmetries to minimize Binary Decision Diagram (BDD) or related representations is well-documented [38, 4, 39, 40, 3]. Much research has demonstrated that a function's symmetry properties may reduce the size of the BDD or related data structure such as Functional Decision Diagrams (FDDs) [37, 39, 41, 42]. In particular, Scholl *et. al.* present a method of BDD minimization based on symmetries [39]. This method is based on heuristics which identify partial symmetries within a function. It is our hypothesis that such heuristics can be expanded to incorporate the antisymmetries that we have defined. This would allow identification of situations where an antisymmetric portion of a function could be shared within the structure of the BDD.

Figure 4.4 shows a 4 variable Boolean function possessing $\overline{E}\{x_3, x_4\}$, and Figure 4.5 shows the Shannon tree for this function. The branches which can be shared due to the anti-equivalence symmetry are indicated by the boxes.

An example of how the antisymmetries in a function reduces the complexity of the function's logic implementation is shown in Figures 4.6 and 4.7. Function $f$, shown

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 1 | 1 |
| 01 | 1 | 0 | 0 | 0 |
| 11 | 1 | 1 | 0 | 1 |
| 10 | 0 | 0 | 1 | 0 |

**Figure 4.4.** *The Karnaugh map for a Boolean function possessing* $\overline{E}\{x_3, x_4\}$.

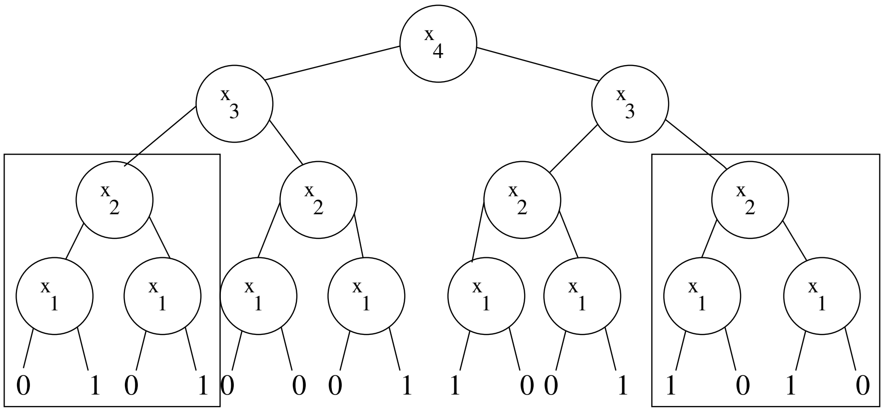


**Figure 4.5.** *A Shannon tree showing two branches which display an anti-equivalence symmetry. Note that the left edge from each node is the 0 edge while the right is the 1 edge.*

on the left in Figures 4.6 and 4.7, possesses the antisymmetry $\overline{S}\{x_4|x_3\}$. Knowing this, we can manipulate the function in such a way that results in a function of a reduced size, plus some additional logic to convert the reduced function into the desired function. The reduced function is shown on the right in Figures 4.6 and 4.7. In this figure, clearly $f(X)$ is a much simpler function than $f^*(X)$ as $f(X)$ requires 4 OR gates and 12 AND gates while $f^*(X)$ requires only 2 OR gates and 5 AND gates (16 vs 7 2-input gates). The advantage of using $f^*$ to implement the function is that there is a greatly reduced number of blocks in the Karnaugh-map. This leads to a smaller number of overall inputs being required, as well as possibly improving

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 1 | 0 |
| 01 | 1 | 1 | 0 | 1 |
| 11 | 0 | 0 | 1 | 1 |
| 10 | 0 | 1 | 0 | 0 |

**a)**

| $x_2x_1 \backslash x_4x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 1 | 1 | 1 | 1 |
| 11 | 0 | 0 | 1 | 1 |
| 10 | 0 | 1 | 0 | 0 |

**b)**

**Figure 4.6.** ***a)*** *the Karnaugh map for* $f(X) = \overline{x}_1\overline{x}_2x_3x_4 \vee x_1\overline{x}_2\overline{x}_4 \vee x_1\overline{x}_3x_4 \vee x_1x_2x_4 \vee \overline{x}_1x_2x_3\overline{x}_4$. ***b)*** *the Karnaugh map for* $f^*(X) = x_1\overline{x}_2 \vee x_1x_4 \vee \overline{x}_1x_2x_3\overline{x}_4$.

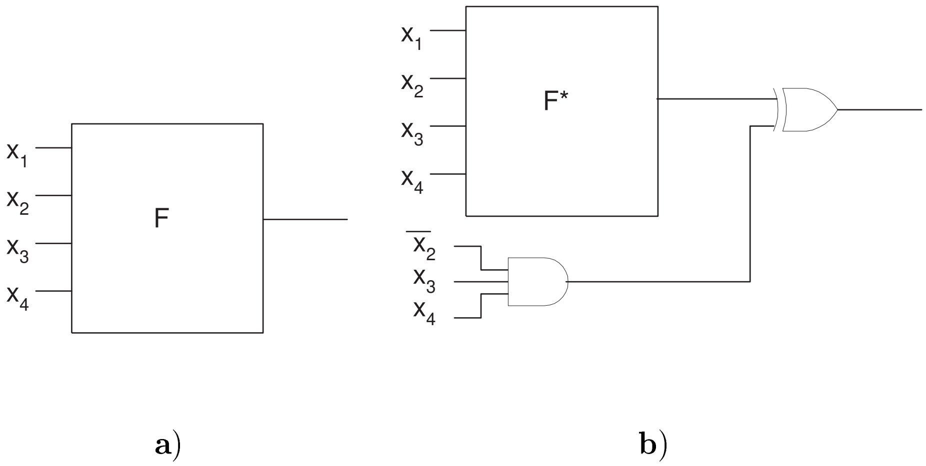


**Figure 4.7.** ***a)*** *Representation of* $f(X)$. ***b)*** *Representation of* $f(X)$ *in terms of a reduced function,* $f^*(X)$.

routing requirements on an FPGA-type implementation. Additionally, $f^*$ clearly has many more symmetries that can be identified, and requires fewer than half of the gates than does the implementation of $f$. Even taking into account the additional logic required to implement $f$ based on $f^*$, there is still a savings of 6 gates.

## 4.8 Conclusion

This chapter discusses in detail the implications that various symmetries have on the values of the autocorrelation coefficients. In particular, theorems describing the resulting autocorrelation coefficient patterns are proven for completely symmetric functions, and for functions possessing equivalence, nonequivalence, and single variable symmetries. During the course of these investigations it was determined that the autocorrelation coefficients alone do not provide enough information to identify the existence of symmetries. This is due to the loss of sign information when computing the autocorrelation coefficients from the spectral coefficients (see Equation 3.1).

However, another contribution of this chapter is the introduction of antisymmetries. This new type of symmetry is defined, and spectral tests and conditions similar to those given for the regular symmetries in [5] are derived. In addition, theorems describing the resulting autocorrelation coefficient patterns are also proven for the six types of antisymmetries. Antisymmetries have clear applications in various types of logic synthesis applications, particularly in the reduction of decision diagrams. Some examples are given to provide some idea of the direction this work may take.

# Chapter 5

# Computation of the Autocorrelation Coefficients

## 5.1 Introduction

$$B(\tau) = \sum_{v=0}^{2^n-1} f(v) \times f(v \oplus \tau) \tag{2.4}$$

The use of Equation 2.4 to compute the autocorrelation coefficients requires on the order of $2^n$ operations to compute each of the $2^n$ coefficients. For even small values of $n$ this has been, until recently, infeasible. In an attempt to solve this problem, techniques that reduce the required numbers of operations have been developed by various researchers. Additionally, as computation resources increase in memory availability and speed it is becoming possible to make use of some of the more compute-intensive techniques. However, a run-time that is exponential in $n$ is still very limiting, and is likely the reason that investigations into the uses and properties of the autocorrelation coefficients have not been previously addressed.

In this chapter we examine implementations of the existing computation techniques and we present two new computation techniques. These new approaches are based on DDs, and make use of the inherent efficiency of this data structure. Other approaches that we have implemented and tested include the straightforward implementations of Equation 2.4 and Equation 3.1, a method based on the function's disjoint cube-list, and techniques for estimating the autocorrelation coefficient values. Each of these approaches have been tested on a variety of benchmark sizes measured in the number of inputs, $n$. One may argue that the use of a single coefficient is limited, and that a technique is only useful if it can compute all $2^n$ coefficients relatively quickly. However, this and previous work [43] has shown that a small subset of the coefficients can be used in various logic synthesis applications. Clearly there are two needs to satisfy: the computation of small numbers of arbitrary coefficients, and the computation of all $2^n$ coefficients. We consequently evaluate the performance of each technique in computing a single coefficient as well as the performance in computing all $2^n$ coefficients. We find that many of the techniques presented in this work are more than fast enough to satisfy the first requirement for even benchmarks with over 100 inputs.

## 5.2 Brute Force

Our investigation into computation techniques began with a straightforward implementation into the autocorrelation function to the switching function's $2^n$ inputs. This use of Equation 2.4 is labeled the "brute force" method. If the equation is directly implemented with no attempt to improve the efficiency, computation of each of the $2^n$ coefficients requires $2^n$ multiplications and $2^n - 1$ addition operations. The advantage of this method is that either a single coefficient or all $2^n$ coefficients may be computed as needed. The only memory requirements are for space to hold the function's truth table (or some reduced representation of it).

### Improvements

There are a few ways that this can be improved upon.

### Computing $B(0)$

The zero-order coefficient, $B(0)$ or $C(0)$, need not take $2^n$-plus operations. $B(0)$ is the number of true minterms $(k)$, and so is a straightforward count. The operations required for this is dependent on the function representation. Computation of $C(0)$ is not required as it is always $2^n$. This saves computation of one of the $2^n$ coefficients.

### Half the Products

Within the summation, it is not strictly necessary to compute $2^n$ products. Since $f(v) = f(v \oplus \tau \oplus \tau)$ it can be shown that only half of the products need be computed. For example,

$$B(2^{n-1}) \quad = 2 \sum_{v=0}^{2^{n-1}-1} f(v) \times f(v \oplus 2^{n-1})$$

Unfortunately, depending on the value of $\tau$, the values over which the summation is performed must be adjusted. This causes a large amount of overhead, rendering the savings of such an approach almost negligible.

### Use only the onset

In practice, the above two improvements do not provide any significant savings. In particular, the overhead required for the second improvement negates any savings that are achieved. However, it is possible to get significant performance improvement if only the onset of the function is considered. In general, functions are described as a list of minterms, usually only the true minterms. If instead of iterating over all $2^n$ minterms, both true and false, one iterates over only the cubes in the onset then clearly the number of operations for each coefficient is dependent on the number of cubes. Note, also that iterating over the *cubes* instead of the true minterms means that if a cube contains many minterms quite significant savings can be achieved.

## 5.3 Wiener-Khinchin Method

In contrast to the Brute Force method, the Weiner-Khinchin (W-K) method requires a significant amount of memory storage space. This is because the Brute Force method may be applied to a representation of the function that consists only of its onset, while the W-K method will always require storage of all $2^n$ intermediate spectral coefficients. However, if there is enough memory, the computation of all $2^n$ autocorrelation coefficients can be performed far faster by this method.

The W-K method makes use of Equation 3.1 ($B = \frac{1}{2^n} \times T^n \times R^2$) to compute the autocorrelation coefficients from the spectral coefficients. Using straightforward matrix multiplication this does not appear to provide any savings over the Brute Force method. However, when a fast transform algorithm is used for the computation of $R$, and then for $B$, $O(n2^n)$ operations are required [5]. This technique requires that all $2^n$ spectral coefficients must be stored, and squared as an intermediate step. Each of these are then used in the computation of each of the $2^n$ autocorrelation coefficients, reusing the results of multiplying individual elements where-ever possible (see Figure 2.12). Because of this, much of the computation savings are lost if we require only one, or even a small subset of the coefficients.

## 5.4 Reuse Method

As discussed in Chapter 3 the $\{+1, -1\}$ autocorrelation coefficients can be computed from the individual sum products of previously computed coefficients. We refer to this method of computing the autocorrelation coefficients as the Reuse method as it is based on the reuse of certain products. However, due to the overhead in determining which products are available for reuse this technique does not appear to offer any significant advantages, and so has not been implemented.

## 5.5 Decision Diagram Methods

As part of this research two new methods for the computation of the autocorrelation coefficients were developed. Each of these methods were based on ROBDDs. The ROBDDs were used as the data structures for storage of the switching function. Both of the methods described below were implemented using the CUDD decision diagram package [44].

### Brute Force BDD

This technique originated as a very straightforward (hence the label "Brute Force") calculation method based on the initial representation of the function as a ROBDD. Computation of the zero-order coefficient is simply a matter of counting the minterms. Computation of higher-order coefficients is performed by creating a ROBDD for the original function and a ROBDD for the function XOR-ed with $\tau$, and then identifying the common minterms. This requires the following steps:

- For each variable with a 1 in the binary expansion of $\tau$, create a new ROBDD by taking the original function and replacing that variable with its negative. Apply the next iteration to the new ROBDD.
- Call the function represented by the new ROBDD $f_2$, and the original $f_1$.
- Perform the AND operation $f_1 \wedge f_2$ and call the result $f_3$.
- Count the minterms in $f_3$.

The first step in this method has the effect of shifting the function by the variable(s) of interest. The second step compares the two functions and results in only minterms that are common in the two functions. The final step counts these minterms, to give us the autocorrelation coefficient. All of the steps are done very quickly due to the efficiency of both the ROBDD data structure and to the fast performance of the CUDD package.

### Recursive

The second DD-based technique developed in this work is a recursive technique that requires only one ROBDD. For each coefficient required, a recursive comparison is done of the appropriate outputs of the function, as represented by the ROBDD. For instance, for the value $\tau = 100$ where the variables for a function $f$ are ordered $x_0x_1x_2$, the algorithm will compare the left subtree of $x_0$ with the right subtree of $x_0$. Where any 1's match in position in the left and right subtrees the value 2 is added to the coefficient being calculated. In this way the reduced properties of the ROBDD are used.

Both of the DD-based techniques are designed primarily for the computation of a single coefficient. Computation of any set of the coefficients requires that the entire process be repeated, and the only savings gained are in the fact that the original ROBDD need only be built once.

## 5.6 Disjoint Cubes Method

A method we refer to as the *Disjoint Cubes method* was introduced in [45]. It uses a list of cubes as the representation of the function. This method takes into account two things: first of all, that not all of the $2^n$ input combinations are in the on-set of the function, and secondly, that a single cube may represent more than one input combination through the use of *don't care* values, meaning that the value of that variable has no effect on the output for the given combination of the other input variables. This technique of computing the autocorrelation coefficients from a list of disjoint cubes is based on a technique for calculating the Rademacher-Walsh coefficients described in [46], and is also very similar to a technique previously published by Varma and Trachtenburg [8, 47]. Briefly, this technique requires a disjoint set of cubes. The algorithm then requires summing the weights, or contributions that each cube makes to the autocorrelation coefficient being computed. See [47] for further details. In [45] the technique was limited to computation of only first order coefficients; however, for the purposes of this dissertation the technique was modified to enable computation of any autocorrelation coefficient. The algorithm followed for computing each coefficient is as follows:

- For the computation of $B(0)$, the contribution of each cube to the value of the coefficient is $2^{d_i}$ where where $d_i$ is the number of don't care values in the $i^{th}$ cube.
- For all other coefficients, the contribution of each cube to coefficient $u$ is $2^{d_i}$ if any variable in a position corresponding to a 1 in the binary expansion of $u$ is a don't care value in the $i^{th}$ cube. If the variable has a 1 or a 0, then the cube must be compared with all the other cubes. For each other cube found where the original cube is the "same" as the other cube with only the variable(s) at positions where $u$ is 1 are changed, then the contribution of the original cube is

$$contribution = contribution + 2 * 2^{d_{ab}}$$

  where $d_{ab}$ is the number of don't cares that are in the same position in both the original cube (cube $a$) and the cube it is being compared to (cube $b$).

The algorithm described in [48] was used in finding the disjoint list of cubes from the input list. Again, this technique is mainly designed for the computation of a single coefficient; it does not take advantage of prior computations in order to save time in latter computations. The only savings that are achieved in computing more than a single coefficient is that the disjoint cube list need only be computed once.

## 5.7 Estimation Methods

Due to the complexity involved in computing the autocorrelation coefficients an alternate approach is to estimate their values, as presented in [49] and [50]. The technique presented in these papers makes use of a technique introduced by Karp *et. al.* [51] for approximating the number of satisfying assignments for a Boolean function. Briefly, the method can be applied to a function in sum-of-products form or product-of-sums form and involves ordering each term (sum or product, depending on the form) arbitrarily. For each term a "random" assignment is determined, and this assignment is then counted if and only if it satisfies no earlier product term (for sum of products) in the ordering, OR if and only if it fails to satisfy no earlier sum term in the ordering (for product of sums). Assuming $n$ variables in the function, of which $j$ are literals in the current term, $s$ total assignments are sampled, and of these, $c$ are counted, then the estimated size of the equivalence class is

$$2^{n-j} \times \frac{c}{s}$$

The estimated number of satisfying assignments for the function is given by the sum of the estimated sizes of the equivalence classes over all the terms in the function. It is shown in [51] that the total number of samples required to guarantee an accuracy of $1 \pm \epsilon$ with a probability of at least $1 - \alpha$ is

$$\frac{4n \ln(\frac{2}{\alpha})}{\epsilon^2}$$

This technique is applied to the problem of estimating the autocorrelation coefficients by creating an expression that represents $f \wedge f^*$, where

$$f^*(v) \triangleq f(v \oplus u)$$

and then estimating the number of satisfying assignments for this expression.

Two versions of this were implemented. The first version uses a straightforward array implementation to store the terms while the second version uses BDDs.

## 5.8 Comparisons

Each of the above techniques for computing the autocorrelation coefficients have been implemented in C++ and tested on a variety of benchmarks. For these tests the benchmarks were preprocessed to ensure that each had only a single output. They were then divided into three classes: benchmarks with 10 or fewer inputs, benchmarks with 11 to 30 inputs, and benchmarks with greater than 30 inputs. This was done because it was clear that some of the techniques would not perform well on large benchmarks, and so in order to generate useful data on their performance at all levels such a division was necessary. Overviews of each implementation and results of these tests are summarized in the following sections.

### 5.8.1 Computation Techniques

**BRUTEFORCE - version 1**

The first computation technique to be implemented is called *bruteforce*, because the algorithm is based directly on the defining equation for the autocorrelation coefficients. The first version of this program is included mainly for comparison purposes.

**BRUTEFORCE - version 2**

The bruteforce program was refined slightly in this version. The main refinement was the use of a BDD [44] as the internal data structure for storing the function. Once again, however, the underlying algorithm is a direct implementation of the defining equation.

**BRUTEFORCE - version 3**

A final refinement of the bruteforce implementation was made for this program. This version of the computation program uses a decision diagram as intermediate storage, as did version 2, but in this case we use only the onset to do the computations instead of iterating through all $2^n$ inputs.

### BDD

The next logical progression is to do all of the computation with a decision diagram. This version of the computation program does exactly that; the original function is stored as a BDD, and for each coefficient, another BDD representing the shifted functions is created and then the number of matching minterms are counted.

### BDD_recursive

BDD_recursive is an alternative autocorrelation computation program that uses BDDs to recursively evaluate the coefficients. In considering this implementation we predicted that the time required to compute a coefficient would increase while the memory requirements for the program should decrease, as only one decision diagram is required.

### DISJOINT CUBES

DISJOINT CUBES is the name of the program utilizing the computation technique based on the disjoint cube list, as described in Section 5.6. The performance of this particular technique clearly will be highly dependent on the number of cubes in the function.

### Wiener-Khinchin

The alternative method for computing the autocorrelation coefficients is to apply a transform matrix twice, with some intermediate computation, as described in section 3.3. The obvious shortcoming of this program is that memory space to store all $2^n$ coefficients is required. The clear advantage of this technique is that computations are shared between coefficients, thus significantly reducing the amount of work required.

**ESTIMATION**

Two versions of the estimation technique have been implemented. Version 1 uses a straightforward list of character strings to store the cubes on which the estimation is performed, while version 2 uses BDDs to store the separate cubes and the initial function. The tests below were run using parameters to specify estimation of the coefficient value within $1 \pm .3$ with a probability of $1 - .02$.

### 5.8.2 Experimental Procedure

To test the various techniques a series of experiments were carried out. An initial series of tests on benchmarks limited to 10 inputs was first performed. The reason for this is that it was clear that the less sophisticated techniques would not perform well with higher numbers of inputs, and so this initial set of tests was run in order to weed out the poorest of the techniques.

The time to compute all $2^n$ coefficients in one run was recorded, and also the time to compute a single coefficient. The reasoning behind this is that it is very likely that only a subset of the coefficients are likely to be used, due to the sheer numbers involved. When computing a single coefficient, it was chosen to compute $B(9)$ for these tests. $B(9)$ was chosen because it represents a typical autocorrelation coefficient and so results from the computation of $B(9)$ can be considered typical for the computation of any coefficient.

The second set of tests involved benchmarks having 11 to 30 inputs. Again, both a single coefficient and all $2^n$ coefficients were computed. A final set of tests involving the computation of only a single coefficient was run on benchmarks from 31 to 140 inputs.

All tests were carried out on 4 machines that were provided by the Canadian Microelectronics Corporation in an equipment loan to the Computer Science and Electrical and Computer Engineering Departments at the University of Victoria. The benchmarks used are in ESPRESSO-MV (or pla) format [52].

### 5.8.3 Results

#### 1 to 10 inputs

Table 5.1 summarizes the timing results for the first set of benchmarks that were tested. The failures column refers to any failure to complete computation of the coefficients. It should be noted that for all tests the averages given are approximate values, and that all required preprocessing such as to compute a disjoint cube set or build a BDD is included in the given figures. Also, in all tests the averages given include only the times for successful completions.

| technique | avg time to compute one coefficient | avg time to compute $2^n$ coefficients | failures |
|---|---|---|---|
| BRUTEFORCE - v1 | 0.78 seconds | 180 seconds | 0 |
| BRUTEFORCE - v2 | 0.06 seconds | 58.1 seconds | 0 |
| BRUTEFORCE - v3 | 0 seconds | 9.76 seconds | 0 |
| BDD | 0.02 seconds | 3.76 seconds | 0 |
| BDD recursive | 0.02 seconds | 4.47 seconds | 0 |
| W-K | 0.32 seconds | 0.54 seconds | 0 |
| DISJOINT | 0.45 seconds | 16.0 seconds | 0 |
| ESTIMATION - v1 | 68.6 seconds | N/A | 0 |
| ESTIMATION - v2 | 96.9 seconds | N/A | 0 |

**Table 5.1.** *Timing results for various autocorrelation computation techniques for benchmarks with 1 to 10 inputs.*

Note that the tests involving computing all $2^n$ coefficients were not run on the estimation techniques. This decision was made due to the fact that these techniques required over a minute to compute a single coefficient. Although this would suggest that use of these techniques for any larger benchmarks would be infeasible, this may not be the case because the computation time is closely linked to the number of products/sums terms in the function. Thus the estimation techniques may be well suited for functions with very large number of inputs but a comparatively small number of products/sums terms. In fact the subsequent tests on larger benchmarks demonstrate that unlike the other techniques the performance of the estimation technique does not increase with $n$.

As expected, the time for most of the methods to compute $2^n$ coefficients is in general considerably larger than the time required for computation of a single coefficient. However, the difference is significantly lower for the Disjoint Cubes method and the W-K method. This is explained by the fact that the algorithm used by the Tranform method is designed to compute all of the coefficients by reusing various internal products and sums. The lower difference in the times for the Disjoint Cubes method is explained by the fact that the determination of the disjoint cube list is quite computationally intensive, and thus in computing more than one coefficient the time for this determination is saved for subsequent computations.

### 11 to 30 inputs

For the second set of experiments, a limit of 30 minutes of CPU time was introduced. In this case, failures may be due to either memory or time limits being reached. 621 benchmarks were included in these tests. Only computation of one coefficient is timed, as the number of coefficients to compute is clearly very large and in most cases requires on the order of $2^n$ times the work required to compute one coefficient. Table 5.2 summarizes these results.

### 31 to 140 inputs

The third set of experiments was run in a similar manner to the previous set; a limit of 30 minutes of CPU time was introduced, and failures were possible when

| **technique** | **avg time to compute one coefficient** | **failures** |
|---|---|---|
| BRUTEFORCE - v3 | 67.6 seconds | 0 |
| BDD | 1.00 seconds | 0 |
| BDD recursive | 1.23 seconds | 0 |
| W-K | 918 seconds | 37 |
| DISJOINT | 8.98 seconds | 0 |
| ESTIMATION - v1 | 328 seconds | 2 |
| ESTIMATION - v2 | 344 seconds | 58 |

**Table 5.2.** *Timing results for various autocorrelation computation techniques for benchmarks with 11 to 30 inputs.*

either memory or time limits were reached. Once again, only the computation for one coefficient was recorded. 964 benchmarks were included in these tests. Table 5.3 summarizes these results.

### Discussion of the Results for Large Benchmarks

**BRUTEFORCE - v3** Using the BRUTEFORCE method only 295 of the 961 files succeeded. Of the 666 failures, 660 of those ran out of CPU time, as expected. The remaining 6 failures were due to problems in the internal data storage (see below).

**BDD methods** Both of the BDD methods succeeded in computing the specified autocorrelation coefficient for 719 of the 961 test files. Of the 242 failures, 236 were due to running out of CPU time. The remaining 6 failures due to running out of memory while building the intial ROBDD for the function. It should be pointed out that these 6 failures occurred on the same 6 files as the non-time-out failures for the BRUTEFORCE method.

| **technique** | **avg time to compute one coefficient** | **successes** |
|---|---|---|
| BRUTEFORCE - v3 | 663 seconds | 295 |
| BDD | 4.58 seconds | 719 |
| BDD recursive | 27.5 seconds | 719 |
| W-K | N/A | 0 |
| DISJOINT | 5.12 seconds | 718 |
| ESTIMATION - v1 | 572 seconds | 713 |
| ESTIMATION - v2 | 30.4 seconds | 718 |

**Table 5.3.** *Timing results for various autocorrelation computation techniques for benchmarks with 31 to 140 inputs.*

**W-K method** The W-K method did not succeed for any of the large input files. Of the failures, only 236 were due to running out of CPU time, leaving 725 failures due to running out of memory.

**DISJOINT CUBES method** The disjoint-cubes method performed extremely well, failing on 243 files due to lack of time and succeeding on all the remaining files in an average of 5 seconds.

**ESTIMATION methods** The two estimation methods had varied performance. The first method, using character arrays to store the cube lists, completed on 713 of the 961 files but requiring an average of over 500 seconds of CPU time. All 248 failures were due to running out of CPU time. The second method, using BDDs to store the cube lists, required an average of 30 seconds to complete for each of the 718 successful benchmark tests. Of the 243 unsucessful tests, 5 were due to being unable to build the BDD while the remainder were due to running over the 30 minute CPU-time limit.

### 5.8.4 Analysis

The goal of these tests is to select a technique for computing the autocorrelation coefficients, given some basic information such as a) the number of coefficients we wish to compute, and b) the size (number of inputs) of the function. Based on these results the best choice for computation of all or one of the coefficients is the first BDD technique developed for this work. The results demonstrate that for both large and medium-sized benchmarks this technique completes with the fewest failures and in the fastest time. The second choice for large benchmarks is the disjoint cubes technique, although for medium-sized benchmarks it does not perform quite as well as the recursive BDD technique. These three options far out-perform all of the remaining techniques for benchmarks having more than 10 inputs.

For 10 or fewer inputs, if all the coefficients are required then the best selection is clearly the W-K method, which always computes all of the coefficients. All but the estimation methods complete computation of one coefficient for a small benchmark in an average time of under one second.

The surprising result is really not that the BDD methods perform so well, or that the W-K method is best for computing all the coefficients for small files; it is that the estimation techniques perform so poorly. The only technique to have an average computation time for large benchmarks that is higher than the best estimation technique was the Brute Force method. As noted above, the estimation parameters specified estimation of each value within $1 \pm .3$ with a probability of $1 - .02$, so although we relaxed the accuracy, we still require a high probability. This indicates that approximately $4000 * n$ samples are required for each coefficient. The numbers, while not that large, are somewhat deceiving, as the algorithm requires that for each of those $4000 * n$ samples they must be checked against the list of products. This leads one to expect a very poor performance for a function with a large number of products, which is likely the case for the larger benchmarks.

## 5.9 Conclusion

During the course of these investigations into the uses of the autocorrelation coefficients we have implemented a number of autocorrelation computation techniques. These techniques are introduced in this chapter, as well as experimental tests for each of the implementations. The implemented techniques include

- two new BDD-based techniques,
- three straight-forward implementations of the autocorrelation function,
- a technique based on the Weiner-Khinchin method for computing the coefficients,
- a technique based on disjoint cube lists, and
- two estimation techniques.

The results of the tests show that the fastest and most successful methods are the new BDD-based techniques developed in this work, with the disjoint cube list technique a close second. The W-K method computes all $2^n$ coefficients very quickly, but runs out of memory on a regular basis for benchmarks with over 10 inputs. The estimation techniques are surprisingly slow given that an exact value is not necessarily computed. The BDD techniques, however, perform very quickly for all sizes of benchmarks. Computation of all $2^n$ coefficients is still infeasible for large values of $n$; however, computation of $n$ coefficients – such as the $n$ first order coefficients – is well within the power of these methods. As shown in Chapters 3, 4 and 7 there are many applications for small subsets of the autocorrelation coefficients. In the worst case, computation of a subset of $s$ coefficients would require $s$ times the computation time for a single coefficient. However, if the initial work in determining the representation – *i.e.* the BDD or disjoint cube list – was not required to be repeated, then the average computation requirements for each coefficient in the subset would clearly be lower than this worst case. This feature has in fact been implemented for each of the computation technniques introduced in this chapter.

It should also be noted that although this work does not present solutions for the computation of autocorrelation coefficientss for multiple-output functions, some discussion of this extension can be presented. In particular, the use of the BDD techniques could be parallelized in order to compute the coefficients for each of the functions represented by the shared BDDs in the same time required for computation of a coefficient for a single-output BDD.

# Chapter 6

# The Autocorrelation Classes

## 6.1 Introduction

As discussed in Chapter 2, there are a number of ways in which switching functions may be grouped into classes. This is useful for a number of reasons:

- it allows analysis to be performed on a representative of each class, the extension of which can lead to a near-optimal implementation for all the members of that class with minimal amounts of processing,
- it allows us to group together functions with similar characteristics, leading to improved logic synthesis techniques tailored to functions with those characteristics, and
- classifying switching functions provides in itself a method of logic synthesis starting from an optimal implementation of a representative function and then adding the necessary additional logic.

As detailed in Chapter 5 we have developed some relatively fast techniques for computation of the autocorrelation coefficients. The feasibility of their computation combined with the interesting properties identified in Chapter 3 indicate that the autocorrelation coefficients have the potential to provide a useful classification technique. Additional motivation comes from the fact that the spectral classification technique is well known and has been widely used. Given the relationship of the autocorrelation coefficients to the spectral coefficients, it seems reasonable that an autocorrelation classification technique could provide many of the useful aspects of the spectral classification technique as well as extending the uses to newer areas of research such as DD representations.

This chapter defines four invariance operations for the autocorrelation classes. Because we have chosen to retain sign information in the autocorrelation classification technique, this technique results in more classes than does the spectral classification technique. The autocorrelation classes are therefore smaller than the spectral classes. Closer examination has indicated that the additional classes are a result of the fact that the sign of the autocorrelation coefficients are affected by the type (v) spectral invariance operation. Thus if one chooses to examine only the magnitudes of the autocorrelation coefficients in an approach similar to that followed in the spectral classification technique, the autocorrelation classes are in fact identical to the spectral classes. This relationship is discussed in Section 6.6.

The benefit of this new classification method is that different information is provided. Chapters 3 and 7 demonstrate that one useful piece of information highlighted by the autocorrelation coefficients allows for the identification of XOR logic within the function, and further work continues on the applications of these classes.

Since much of the motivation behind determining classes of similar switching functions has focused on the advantage of having a representative of each class, we also define a canonical representative of each autocorrelation class, in terms of the autocorrelation spectra. Our classification technique lends itself to many practical applications, and we propose a small number of these in Section 6.7.

## 6.2 Definition

There are a number of operations which, when applied to a switching function, result in a new function whose autocorrelation coefficients are unchanged in values from the original. Their ordering may be modified, but the values remain the same. These are referred to as autocorrelation invariance operations. The autocorrelation classes are defined by grouping together all switching functions whose autcorrelation coefficients remain unchanged after applying one or more of the autocorrelation invariance operations.

The four autocorrelation invariance operations consist of

- permutation,
- input negation,
- replacement of an input $x_i$ with $x_i \oplus x_j$, $i \neq j$ (the type (iv) spectral invariance operation[1]), and
- output negation.

As is discussed below, output negation affects the $\{0, 1\}$ and $\{+1, -1\}$ coefficients quite differently. For the purpose of defining the autocorrelation classes we use only the $\{+1, -1\}$ autocorrelation coefficients. For proving the results of the invariance operations the choice of $\{+1, -1\}$ or $\{0, 1\}$ encoding is made according to whichever leads to the simpler exposition of the proof.

Each class defined by these operations may be related to one or more other class through the application of the type (v) spectral invariance operation. Identification of these related classes is relatively easy, since the only difference in autocorrelation values between the classes is the sign of the values.

## 6.3 Invariance Operations

In this section we formally define each of the invariance operations and their resulting effects on the autocorrelation coefficients.

[1]The numeric labeling of each of these types of operations is taken from [5], in which the spectral invariant operations are labeled (i) through (v)

### 6.3.1 Permutation

The first invariance operation for the autocorrelation classes is permutation of the input variables. Since the autocorrelation coefficients are directly related to the structure of the function, any relabeling of the variables is directly reflected in the labeling of the coefficients. For this and many of the theorems below it is necessary to distinguish between the autocorrelation coefficients for two functions $f$ and $f^*$. Therefore a single superscript is used to indicate for which function the coefficient is being computed.

**Theorem 6.1** *If* $f(x_n, ..., x_j, ..., x_k, ..., x_1) = f^*(x_n, ..., x_k, ..., x_j, ..., x_1)$, $j, k \in \{1, ..., n\}$ *and* $j \neq k$ *then*

$$B^{f^*}(\tau_{j\alpha}) = B^f(\tau_{k\alpha})$$

while

$$B^{f^*}(\tau_\alpha) = B^f(\tau_\alpha)$$

$\forall\ \alpha$ *such that* $j, k \notin \alpha$.

**Proof.**

Permutation can be described as simply relabeling the inputs. The explanation given by Hurst [11] can be extended to the autocorrelation coefficients as follows. Without loss of generality let us define two functions
$f^*(x_n, x_{n-1}, ..., x_1) = f(x_{n-1}, x_n, ..., x_1)$. In this proof we use the ranges $A, B, C$, and $D$ and the associated notation $f(A), f(B), f(C)$, and $f(D)$ as defined in section 4.3. Then

$$v \ \in \ A \Rightarrow v \oplus \tau_{n\ \alpha} \ \in \ C,$$

$$v \ \in \ B \Rightarrow v \oplus \tau_{n\ \alpha} \ \in \ D,$$

$$v \ \in \ C \Rightarrow v \oplus \tau_{n\ \alpha} \ \in \ A, \text{ and}$$

$$v \ \in \ D \Rightarrow v \oplus \tau_{n\ \alpha} \ \in \ B$$

and

$$\begin{aligned}
v &\in A \Rightarrow v \oplus \tau_{n-1\ \alpha} \in B, \\
v &\in B \Rightarrow v \oplus \tau_{n-1\ \alpha} \in A, \\
v &\in C \Rightarrow v \oplus \tau_{n-1\ \alpha} \in D, \text{ and} \\
v &\in D \Rightarrow v \oplus \tau_{n-1\ \alpha} \in C.
\end{aligned}$$

which gives

$$\begin{aligned}
B^f(\tau_{n\ \alpha}) &= \sum_{v\ \in\ A} f(A) \times f(C) + \sum_{v\ \in\ B} f(B) \times f(D) \\
&\quad + \sum_{v\ \in\ C} f(C) \times f(A) + \sum_{v\ \in\ D} f(D) \times f(B) \\
&= 2\sum_{v\ \in\ A} f(A) \times f(C) + 2\sum_{v\ \in\ B} f(B) \times f(D)
\end{aligned}$$

and

$$\begin{aligned}
B^{f^*}(\tau_{n-1\ \alpha}) &= \sum_{v\ \in\ A} f^*(A) \times f^*(B) + \sum_{v\ \in\ B} f^*(B) \times f^*(A) \\
&\quad + \sum_{v\ \in\ C} f^*(C) \times f^*(D) + \sum_{v\ \in\ D} f^*(D) \times f^*(C) \\
&= 2\sum_{v\ \in\ A} f^*(A) \times f^*(B) + 2\sum_{v\ \in\ B} f^*(C) \times f^*(D).
\end{aligned}$$

By definition, $f(A) = f^*(A)$, $f(B) = f^*(C)$, $f(C) = f^*(B)$ and $f(D) = f^*(D)$ and so

$$\begin{aligned}
B(\tau_{n\ \alpha}) &= 2\sum_{v\ \in\ A} f^*(A) \times f^*(B) + 2\sum_{v\ \in\ B} f^*(C) \times f^*(D) \\
&= B^{f^*}(\tau_{n-1\ \alpha}).
\end{aligned}$$

Clearly the reverse, $B^f(\tau_{n-1\ \alpha}) = B^{f^*}(\tau_{n\ \alpha})$ is also true by the same process.

■

### 6.3.2 Input Negation

The second invariance operation for the autocorrelation classes is negation of any of the inputs.

**Theorem 6.2** *If* $f^*(x_n, ..., x_i, ...x_1) = f(x_n, ..., \overline{x}_i, ..., x_1),\ i \in \{1, ..., n\}$ *then*

$$B^{f^*}(\tau) = B^f(\tau)\ \forall\ \tau.$$

**Proof.**

Without loss of generality let us define $f^*(x_n, ..., x_1) = f(\overline{x}_n, ..., x_1)$. Then by definition

$$\sum_{v=0}^{2^{n-1}-1} f^*(v) = \sum_{v=2^{n-1}}^{2^n-1} f(v)$$

and

$$\sum_{v=2^{n-1}}^{2^n-1} f^*(v) = \sum_{v=0}^{2^{n-1}-1} f(v)$$

and so

$$\begin{aligned} B^{f^*}(\tau) &= \sum_{v=0}^{2^{n-1}-1} f^*(v) \times f^*(v \oplus \tau) + \sum_{v=2^{n-1}}^{2^n-1} f^*(v) \times f^*(v \oplus \tau) \\ &= \sum_{v=2^{n-1}}^{2^n-1} f(v) \times f(v \oplus \tau) + \sum_{v=0}^{2^{n-1}-1} f(v) \times f(v \oplus \tau) \\ &= B^f(\tau) \end{aligned}$$

■

### 6.3.3 Exclusive-or with Input

This operation is generally referred to the type (iv) operation, since it is the fourth type of invariance operation for the spectral classes. This operation, replacement of an input $x_i$ with $x_i \oplus x_j$, is also an invariance operation for the autocorrelation classes.

**Theorem 6.3** *If* $f^*(x_n, ..., x_i, ..., x_j, ..., x_1) = f(x_n, ..., x_i \oplus x_j, ..., x_j, ..., x_1),\ i, j \in \{1, ..., n\}$, *and* $i \neq j$ *then*

$$B^{f^*}(\tau_{\bar{i}j\alpha}) = B^f(\tau_{ij\alpha})$$

while

$$B^{f^*}(\tau_\alpha) = B^f(\tau_\alpha)$$

$\forall\ \alpha$ *such that* $i, j \notin \alpha$.

**Explanation**

If $x_i$ is replaced with $x_i \oplus x_j$ then the coefficients associated with $x_i$ are not affected, while the coefficients associated with $x_j$ are exchanged with those associated with both $x_i$ and $x_j$.

**Proof.**

Without loss of generality, let us define two functions $f^*(x_n, x_{n-1}, ..., x_1) = f(x_n, x_{n-1} \oplus x_n, ..., x_1)$. Then, again using the ranges $A...D$ and notation $f(A)...f(D)$

$v \in A \Rightarrow v \oplus \tau_{n\ n-1\ \alpha} \in D$,

$v \in B \Rightarrow v \oplus \tau_{n\ n-1\ \alpha} \in C$,

$v \in C \Rightarrow v \oplus \tau_{n\ n-1\ \alpha} \in B$ and

$v \in D \Rightarrow v \oplus \tau_{n\ n-1\ \alpha} \in D$

while

$v \in A \Rightarrow v \oplus \tau_{n\ \overline{n-1}\ \alpha} \in C$,

$v \in B \Rightarrow v \oplus \tau_{n\ \overline{n-1}\ \alpha} \in D$,

$v \in C \Rightarrow v \oplus \tau_{n\ \overline{n-1}\ \alpha} \in A$ and

$v \in D \Rightarrow v \oplus \tau_{n\ \overline{n-1}\ \alpha} \in B$.

Then

$$
\begin{aligned}
B^f(\tau_{n\ n-1\ \alpha}) &= \sum_{v\ \in\ A} f(A)\times f(D) + \sum_{v\ \in\ B} f(B)\times f(C) \\
&\quad + \sum_{v\ \in\ C} f(C)\times f(B) + \sum_{v\ \in\ D} f(D)\times f(A) \\
&= 2\sum_{v\ \in\ A} f(A)\times f(D) + 2\sum_{v\ \in\ B} f(B)\times f(C)
\end{aligned}
$$

and

$$
\begin{aligned}
B^{f^*}(\tau_{n\ \overline{n-1}\ \alpha}) &= \sum_{v\ \in\ A} f^*(A)\times f^*(C) + \sum_{v\ \in\ B} f^*(B)\times f^*(D) \\
&\quad + \sum_{v\ \in\ C} f^*(C)\times f^*(A) + \sum_{v\ \in\ D} f^*(D)\times f^*(B) \\
&= 2\sum_{v\ \in\ A} f^*(A)\times f^*(C) + 2\sum_{v\ \in\ B} f^*(B)\times f^*(D).
\end{aligned}
$$

By definition $f^*(A) = f(A)$, $f^*(B) = f(B)$, $f^*(C) = f(D)$ and $f^*(D) = f(C)$ giving

$$
\begin{aligned}
B^{f^*}(\tau_{n\ \overline{n-1}\ \alpha}) &= 2\sum_{v\ \in\ A} f(A)\times f(D) + 2\sum_{v\ \in\ B} f(B)\times f(C) \\
&= B^f(\tau_{n\ n-1\ \alpha}).
\end{aligned}
$$

■

### 6.3.4 Output Negation

Output negation is also an invariance operation for the autocorrelation classes, but only if $\{+1, -1\}$ notation is used.

**Theorem 6.4** *If $g^*(x) = \overline{g(x)}$ then*

$$C^{g^*}(\tau) = C^g(\tau) \ \ \forall\ \tau.$$

**Proof.**

Let us define two functions $g^*(x) = \overline{g(x)}$. In $\{+1, -1\}$ encoding the arithmetic equivalent is $g^*(x) = -g(x)$. Therefore

$$\begin{aligned} C^{g^*}(\tau) &= \sum_{v=0}^{2^n-1} g^*(v) \times g^*(v \oplus \tau) \\ &= \sum_{v=0}^{2^n-1} - g(v) \times -g(v \oplus \tau) \\ &= \sum_{v=0}^{2^n-1} g(v) \times g(v \oplus \tau) \\ &= C^g(\tau) \end{aligned}$$

■

**Corollary 6.4.1** *If* $f^*(x) = \overline{f(x)}$ *then*

$$B^{f^*}(\tau) = B^f(\tau) - 2k + 2^n.$$

**Proof.** In $\{0,1\}$ notation let us similarly define two functions $f^*(x) = \overline{f(x)}$. The arithmetic equivalent is $f^*(x) = 1 - f(x)$. Then

$$
\begin{aligned}
B^{f^*}(\tau) &= \sum_{v=0}^{2^n-1} f^*(v) \times f^*(v \oplus \tau) \\
&= \sum_{v=0}^{2^n-1} (1 - f(v)) \times (1 - f(v \oplus \tau)) \\
&= \sum_{v=0}^{2^n-1} (\ f(v) \times f(v \oplus \tau) - f(v) - f(v \oplus \tau) + 1\ ) \\
&= B^f(\tau) - \sum_{v=0}^{2^n-1} f(v) - \sum_{v=0}^{2^n-1} f(v \oplus \tau) + \sum_{v=0}^{2^n-1} 1 \\
&= B^f(\tau) - 2k + 2^n.
\end{aligned}
$$

■

## 6.4 Spectral Invariance Operations & their Effect on the AC Classes

While the type (v) spectral invariance operation has significant impact for the spectral classes, the autocorrelation (AC) classes are not invariant under its application. As shown below, this operation results in the modification of the signs of the affected autocorrelation coefficients.

Two autocorrelation coefficients with values $2^n$ and $-2^n$ indicate very different structures within a function; one indicates the highest possible degree of similarity, and one indicates the least possible degree of similarity. For this reason it was decided that retention of the sign information within the autocorrelation classification technique is essential. However, it is clear that this fifth spectral invariance operation has some significance for the autocorrelation classes, if only in the fact that its application results in classes identical to those resulting from the spectral classification technique.

## Exclusive-or with Output

Like output negation, the type (v) spectral invariance operation has a very different result on the autocorrelation coefficients depending on which encoding has been chosen. The type (v) spectral invariance operation involves combining the function's output with one of the inputs using the XOR operator.

**Theorem 6.5** *If $f^*(x) = f(x) \oplus x_i$, $i \in \{1, ..., n\}$, then*

$$C^{f^*}(\tau_{\ i\alpha}) = -C^f(\tau_{\ i\alpha})$$

*and*

$$C^{f^*}(\tau_{\ \bar{i}\alpha}) = C^f(\tau_{\ \bar{i}\alpha})$$

$\forall\ \alpha$ *such that* $i \notin \alpha$.

**Proof.**

Without loss of generality let us define two functions $g^*(x) = g(x) \oplus x_n$. If we define two ranges as follows

$A = 0\ ..\ 2^{n-1} - 1$ and

$B = 2^{n-1}\ ..\ 2^n - 1,$

then

$v\ \in\ A \Rightarrow v \oplus \tau_{\ n\alpha}\ \in\ B,$

$v\ \in\ B \Rightarrow v \oplus \tau_{\ n\alpha}\ \in\ A$

and

$v\ \in\ A \Rightarrow v \oplus \tau_{\ \overline{n}\alpha}\ \in\ A,$ and

$v \in B \Rightarrow v \oplus \tau_{\overline{n}\alpha} \in B.$

We can then write the autocorrelation computations as

$$C^{g^*}(\tau_{n\alpha}) = 2 \sum_{v \in A} g^*(A) \times g^*(B)$$

and

$$C^{g}(\tau_{n\alpha}) = 2 \sum_{v \in A} g(A) \times g(B).$$

By definition, $g^*(A) = g(A)$ and $g^*(B) = -g(B)$ and so

$$\begin{aligned} C^{g}(\tau_{n\alpha}) &= 2 \sum_{v \in A} g^*(A) \times (-g^*(B)) \\ &= -C^{g^*}(\tau_{n\alpha}). \end{aligned}$$

Similarly,

$$C^{g^*}(\tau_{\overline{n}\alpha}) = \sum_{v \in A} g^*(A) \times g^*(A) + \sum_{v \in B} g^*(B) \times g^*(B)$$

and

$$\begin{aligned} C^{g}(\tau_{\overline{n}\alpha}) &= \sum_{v \in A} g(A) \times g(A) + \sum_{v \in B} g(B) \times g(B) \\ &= \sum_{v \in A} g^*(A) \times g^*(A) + \sum_{v \in B} (-g^*(B)) \times (-g^*(B)) \\ &= C^{g^*}(\tau_{\overline{n}\alpha}). \end{aligned}$$

■

## Effect on the Autocorrelation Classes

As demonstrated in the previous section, the effect of the type (v) spectral invariance operation (assuming $\{+1, -1\}$ encoding) is to negate certain coefficients. Thus certain of the autocorrelation classes are related to each other through the application of this operation. These related classes are easily identifiable, as they have the same magnitudes of autocorrelation coefficients, but different signs for all or some of the coefficients.

## 6.5 Canonical Autocorrelation Spectra

The canonical spectrum for the spectral classes consists of a list of the $2^n$ coefficient values in the identification order $s_0, s_1, s_2, ..., s_n, s_{12}, ..., s_{12...n}$ but with the zero- and first-order coefficients all positive integer values and arranged in decreasing magnitude. Since the sum of the spectral coefficients must always total $\pm 2^n$ this implies a relationship amongst all the coefficients, and so modifying the spectra to match the given arrangement for the lower order coefficients will clearly affect the higher order coefficients [5]. This choice of canonical representation was made in order to ensure that the canonic function for each spectral class is predominantly first ordered; that is, that the first-order coefficients have the highest magnitude over all the spectral coefficients of the function. According to [12] this ensures that the representative function has an optimum synthesis in terms of threshold logic.

For the autocorrelation coefficients it would seem logical to select a canonic representation with similar underlying reasoning. High values in the first-order spectral coefficients imply a certain degree of simplicity in the function as they indicate similarity between the function and the single-variable functions $f(X) = x_i$ for $i \in \{1, ..., n\}$. High values in the first-order spectral coefficients also correspond to high values in the first-order autocorrelation coefficients and so choosing a canonical representation with this as a requirement ensures that the chosen representative is as simple as can be achieved. In order to maintain a degree of consistency with the spectral representatives, the autocorrelation representatives are chosen such that the magnitude of values are decreasing over the $\{+1, -1\}$ first-order coefficients $C(\tau_n), C(\tau_{n-1}), ..., C(\tau_1)$. We further expand on this choice in the following section.

$$\frac{1}{2^n} \times \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix} \times \begin{bmatrix} s_0^2 \\ s_1^2 \\ s_2^2 \\ s_{12}^2 \\ s_3^2 \\ s_{13}^2 \\ s_{23}^2 \\ s_{123}^2 \end{bmatrix} = \begin{bmatrix} C(0) \\ C(\tau_1) \\ C(\tau_2) \\ C(\tau_{12}) \\ C(\tau_3) \\ C(\tau_{13}) \\ C(\tau_{23}) \\ C(\tau_{123}) \end{bmatrix}$$

**Figure 6.1.** *A three variable example of computing the autocorrelation coefficients from the spectral coefficients using the Hadamard transform matrix.*

## Derivation of Canonical Autocorrelation Representation

The canonical ordering for the representatives of the autocorrelation classes is derived from that used for the spectral classes. In Chapter 3, Equation 3.1 ($\frac{1}{2^n} \times T^n \times S^2 = C$) defines how the spectral coefficients may be used to compute the autocorrelation coefficients. Figure 6.1 demonstrates how each spectral coefficient contributes to each autocorrelation coefficient in a three variable example.

If the first-order spectral coefficients are arranged in decreasing order of magnitude for $s_0, s_1, ..., s_n$, one can see that the autocorrelation coefficient with the most contribution from the largest of these spectral coefficients is $C(0)$, followed by $C(\tau_3)$ which has positive contributions from $s_0$, $s_1$ and $s_2$, followed by $C(\tau_2)$ which has positive contributions from $s_0$, $s_1$ and $s_3$, and so on. Due to the recursive nature of the chosen transform matrix (the Hadamard matrix), this observation can be extended to any number of variables. For these reasons the canonical representative for each autocorrelation class is chosen to be the function with the largest values in the first order coefficients $C(0), C(\tau_n), C(\tau_{n-1}), ..., C(\tau_1)$ followed by the second order coeffi-

| fn no. | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| 2 | 16 | 16 | 16 | 16 | -16 | 16 | 16 | -16 | 16 | -16 | -16 | -16 | -16 | -16 | 16 | -16 |
| 3 | 16 | 16 | 16 | 0 | 0 | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 16 | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | -16 | 0 | 0 | 0 | -16 |
| 5 | 16 | 16 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| 6 | 16 | 16 | 8 | 8 | -8 | 8 | 8 | -8 | 8 | -8 | -8 | -8 | -8 | -8 | 8 | -8 |
| 7 | 16 | 12 | 12 | 12 | -12 | 12 | 12 | -12 | 12 | -12 | -12 | -12 | -12 | -12 | 12 | -12 |
| 8 | 16 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 9 | 16 | 12 | 12 | 4 | 4 | 12 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 10 | 16 | 12 | 12 | 4 | -4 | 12 | 4 | -4 | 4 | -4 | -4 | -4 | -4 | -4 | 4 | -4 |
| 11 | 16 | 12 | 4 | 4 | -4 | 4 | 4 | -4 | 4 | -4 | -4 | -12 | -4 | -4 | 4 | -12 |
| 12 | 16 | 8 | 8 | 8 | 0 | 0 | 8 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 8 | 0 |
| 13 | 16 | 8 | 8 | 8 | -8 | 8 | 8 | -8 | 8 | -8 | -8 | -8 | -8 | -8 | 8 | -16 |
| 14 | 16 | 8 | 8 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | -8 | -8 | 0 | 0 | -8 |
| 15 | 16 | 8 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | -8 | -8 | -8 | 0 | 0 | -8 |
| 16 | 16 | 4 | 4 | 4 | 4 | 4 | 4 | -4 | -4 | 4 | -4 | -4 | -4 | 4 | 4 | 4 |
| 17 | 16 | 4 | 4 | 4 | 4 | 4 | -4 | -4 | -4 | -4 | 4 | -4 | -4 | -4 | -4 | -4 |
| 18 | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $\tau$ | 0000 | 1000 | 0100 | 0010 | 0001 | 1100 | 1010 | 1001 | 0110 | 0101 | 0011 | 0111 | 1011 | 1101 | 1110 | 1111 |

**Table 6.1.** *The canonical representatives for the $n \leq 4$ autocorrelation classes in $\{+1, -1\}$ notation.*

cients $C(\tau_{n\ n-1}), C(\tau_{n\ n-2}), ..., C(\tau_{21})$ followed by the third order coefficients and so on.

Table 6.1 lists the canonical autocorrelation class representatives as selected based on the above criteria for $n \leq 4$.

## 6.6 The Relationship Between the AC & Spectral Classes

There is clearly a tight coupling between the spectral autocorrelation classes, as the invariance operations defining each classification are similar. Given this information, if two functions both in the same autocorrelation class, does this imply that they are both in the same spectral class and vice versa?

Let us define two functions $f_1(X)$ and $f_2(X)$ such that $f_1(X)$ and $f_2(X)$ are in the same autocorrelation class. Then by definition we know that either

$$f_1(X) = f_2(X^*)$$

or

$$f_1(X) = \overline{f_2(X^*)}$$

where $X^*$ represents the inputs $X$ modified by one of the three autocorrelation invariance operations that affect the inputs. The last invariance operation affects the output, and is negation, hence the two options given above.

We know that $f_2(X)$ and $\overline{f_2(X)}$ are in the same spectral class, as output negation is one of the spectral invariance operations. We also know that all three of the remaining autocorrelation invariance operations are also spectral invariance operations, so then if $f_1(X) = f_2(X^*)$ or $f_1(X) = \overline{f_2(X^*)}$ then they must be in the same spectral class, by definition.

Now let us redefine our functions such that $f_1(X)$ and $f_2(X)$ are known to be in the same spectral class. Then $f_1(X) = f_2^{**}(X^*)$ where $**$ represents a type (iii) (output negation) or type (v) (replacement of the output with the exclusive-or combination of the output and an input), and $*$ represents a type (i) (permutation), type (ii) (input negation) or type (iv) (replacement of an input with the exclusive-or combination of that input and another input) spectral invariance operation. Then either the two functions are in the same autocorrelation class (if the type (v) invariance operation is not used) or they are in a different class. If they are in a different class, then we can narrow down which classes they may belong to, as certain autocorrelation classes are related to each other by the type (v) transformation, as discussed in Section 6.4.

This further implies that if the type (v) operation is also applied and one examines only the *magnitude* of the resulting autocorrelation coefficients, then the resulting smaller set of classes is identical to the spectral classes.

## 6.7 Applications of the Autocorrelation Classes

There are various ways in which the classification technique introduced in this chapter may be used in logic synthesis applications.

### Two-Level Logic Synthesis

The spectral classes have been of particular value in logic synthesis. They allow functions in the same spectral class to be synthesized from the class' canonical function by adding logic to complement and/or permute the variables, and/or by appending suitable XOR logic. The same can be applied using the autocorrelation classes. For example, the function $f(X) = x_1x_2 \vee x_1x_3 \vee x_2x_3$ is a very desirable function, as it is totally symmetric. Given the autocorrelation coefficients for $f$ and $f^*$, some function in the same autocorrelation class as $f$, appropriate logic may be added to convert this desirable function – which is likely to have an efficient implementation due to the symmetry property it possesses – into $f^*$, which may or may not possess the same property.

Figure 6.3 illustrates such a situation, given the functions $f$ and $f^*$ as defined in Figure 6.2.

The problem with this application is that one of the autocorrelation invariance operations, input negation, has no effect on the autocorrelation coefficients. With the other invariance operations, examination of the changes in the coefficients will lead to a determination of which operations have been applied to the initial function. This is not so with input negation. Work developing an algorithm for this process must take this factor into account, either disregarding whether positive or negative literals are used in the functions, or by performing some sort of verification at each stage of the algorithm to determine if the correct function has been synthesized.

| $x_1x_2x_3$ | $f(X)$ | B | C | $f^*(X)$ | B | C |
|---|---|---|---|---|---|---|
| 000 | 0 | 4 | 8 | 1 | 4 | 8 |
| 001 | 0 | 2 | 0 | 1 | 2 | 0 |
| 010 | 0 | 2 | 0 | 0 | 2 | 0 |
| 011 | 1 | 2 | 0 | 1 | 2 | 0 |
| 100 | 0 | 2 | 0 | 1 | 2 | 0 |
| 101 | 1 | 2 | 0 | 0 | 2 | 0 |
| 110 | 1 | 2 | 0 | 0 | 0 | -8 |
| 111 | 1 | 0 | -8 | 0 | 2 | 0 |

**Figure 6.2.** *The truth table and autocorrelation vectors for two functions in the same autocorrelation class.*

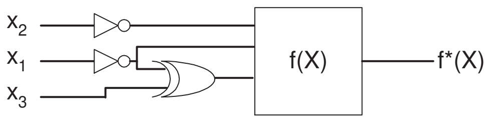


**Figure 6.3.** *The additional logic required to convert $f$ into $f^*$.*

## Decision Diagrams

The use of the autocorrelation classes can also be demonstrated in relation to logic synthesis involving decision diagrams.

Given two functions $f(X)$ and $g(X)$ in the same autocorrelation class, a straightforward technique for transforming a BDD representation for $f$ to $g$ (or vice versa) can be determined. Initial investigation in this area indicates that the application of any of the first three autocorrelation invariant operations to the BDD representation of a function is trivial: permutation of two variables is easily performed simply by relabeling the nodes in the graph, and negation of either the output or inputs of a function can be performed through the use of inverters. The final invariant operation requires a more complex solution, and has the potential to cause changes in the size of the BDD (unlike the previous operations). In practice, replacement of an input variable $x_i$ with $x_i \oplus x_j$ requires only the addition of two internal inverters. These inverters are used to invert the nodes on the branches where $x_j$ has the assignment 1 (assuming $x_i$ follows $x_j$ in the current variable ordering). This is not a difficult operation to perform, but may cause the BDD to grow as previously isomorphic (and thus shared) sub-graphs may no longer be isomorphic. Additions of inverters or the use of a KDD in place of the BDD would certainly solve this problem.

Further work is continuing in this area. Assuming that a "good" decision diagram representation is one that has a size (*i.e.* the number of nodes) that is not exponential in the number of inputs, then the goal is to construct a fast algorithm for generating a good decision diagram for $f$ given a good decision diagram for $g$, and assuming that $f$ and $g$ are both in the same autocorrelation class.

### BDD vs FDD vs KDD Classes

If canonical representatives of the $n \leq 4$ autocorrelation classes are examined, it is possible to quickly identify certain functions which contain the markers indicating XOR logic (first order coefficients with the value $-2^n$ or second order coefficients such that $C(\tau) = 0$, $\tau \in \{x_1, ..., x_i, x_j, ..., x_n\}$ where $x_k = 0 \; \forall \; k \neq i, j$ and $x_i x_j \in \{01, 10, 11\}$). In particular, from Table 6.1 function number 4 contains the second type of marker, and functions number 2 and 18 contain the first type of marker.

Other function numbers such as 3, 12, 14 and 15 do not exhibit the required patterns in the canonical function, but have the potential through the application of invariance operations to contain the XOR-identifying patterns. This leads one to speculate whether the autocorrelation classes could provide an immediate test as to whether a function should be represented with a BDD, a FDD, or a KDD. This is related to the above problem of how to represent a function after the application of the invariant operation $x_i \leftarrow x_i \oplus x_j$. Again, further work is continuing in this area.

## Design for Embedded Systems

An area of great interest for many years has been the design of a universal function. The concept involves determining one function from which, through the application of various simple operations, any other function may be implemented. This has so far turned out to be an infeasible goal. The introduction of the new classification presented in this work, combined with a new technology allowing reconfigurable hardware allows us to revisit this concept.

One common reconfigurable hardware device is a FPGA stands for *Field Programmable Gate Array*. As the name suggests, a FPGA is an array of devices whose interconnects can be programmed by a user without the use of large and expensive equipment. A more formal definition for the term FPGA can be stated as [53]:

> "A FPGA is a device in which the final logic structure can be directly configured by the end user, without the use of an integrated circuit fabrication facility."

Generally, the designer uses a set of CAD tools to minimize the function to be implemented, and then downloads the optimized function to the FPGA.

The classification technique introduced in Chapter 6 differs from other techniques in that it groups together functions with significant amounts of similarity within their structures. It may be possible to implement a canonical function on a FPGA and then add extraneous logic to create the desired function from that class. Optimization would then be performed for each of the canonical function implementations. Having a pre-optimized description of each canonical function would save considerably on the pre-processing and downloading overhead, which can be considerable. This technique could be approached in a number of ways:

- One could take the optimized description for the canonical function and add the required logic, then download the function to the FPGA; or
- the optimized function could be downloaded to the FPGA followed by the down-

load of the additional logic; or

- two FPGAs could be used, one for the optimized function and one for the additional logic.

The second and third techniques would allow for minimal reprogramming overhead should a new function of the same class be required. New techniques are currently being developed to allow partial reconfiguration in the manner described in the second technique [54].

## 6.8 Conclusion

Classification techniques have historically proven very useful in logic synthesis applications. There is new interest in logic synthesis techniques such as three-level minimization and XOR-based DD representations, and we have seen in Chapter 3 that the autocorrelation coefficients may be used to identify properties of use in these applications. Based on this we have hypothesized that the autocorrelation coefficients provide a classification method that is of use in these recently introduced areas.

In this chapter we define the autocorrelation classes based on four autocorrelation invariance operations, and we prove the effects of these operations on the coefficients. We also discuss the relationship between the well-known spectral classes and our newly defined autocorrelation classes. The chapter goes on to define canonical representations for each class, and the reasons for our choice in this matter. There are many areas in which our classification technique may be applied, and a number of these are addressed in Section 6.7.

# Chapter 7

# Applications

## 7.1 Introduction

This dissertation has so far discussed mainly theoretical aspects of the autocorrelation function. Chapter 6 discusses potential uses for the autocorrelation classes, but no implementations or tests are performed. In this chapter we discuss in detail some applications of the previously introduced concepts. In particular, we are interested in applying Theorems 3.8 and 3.9 from Chapter 3. These are elaborated on in Section 7.2.

The majority of this chapter is devoted to the discussion of two implementations that make use of the properties from Theorems 3.8 and 3.9. In Section 7.3 a three-level decomposition tool is presented, while Section 7.4 presents a tool for determining decomposition type lists for KDDs. These implementations are contrasted and compared with existing tools for each application. The results from both are very promising, particularly given that the implemented algorithms are quite straight-forward, and that the results are in comparison with tools representing many years of work.

## 7.2 Identification of Exclusive-OR (XOR) Logic

In many cases, functions with embedded XOR logic have very large two-level expressions such as sum-of-products and product-of-sums expressions [55]. In many of these cases, the choice of a representation that includes XOR logic can significantly reduce the size. For example, an $n$-variable parity function requires $2^{n-1}$ product terms each consisting of $n$ literals if expressed using AND, OR and NOT. However, if XOR logic is included, then only $n$ single-literal product terms are required. The problem is that minimizing a function for only two levels of operators is approached quite differently than is three-level minimization. Clearly the identification of the presence of XOR logic before attempting any further minimization is a useful thing.

BDDs are also quite inefficient for representing certain types of functions such as multipliers and those containing XOR logic [13, 40]. Becker *et. al.* demonstrate in [56] that there exists a class of functions that cannot be represented efficiently by OBDDs but can be by OFDDs. They also show that there are functions for which both OBDD and OFDD representations are exponential in size, but for which polynomial-sized OKFDDs (KDDs) may be constructed. The problems consist of

1. identifying the functions that do not have polynomial-sized BDD representations, and then
2. identifying the decompositions that will lead to a polynomial-sized KDD.

Again, the usefulness of identifying the existence and location of XOR logic within a function is clearly of use.

Section 3.5.4 in Chapter 3 shows how the autocorrelation coefficients may be used in identifying various forms of XOR logic. Briefly, a first order coefficient with the value $-2^n$ indicates that the function can be decomposed into $f(X) = f^*(X) \oplus x_i$, and additional examination of the second order coefficients may lead to the identification of a decomposition of the form $f(X) = f^*(X) \oplus (x_i * x_j)$. Section 7.3 discusses the application of these properties to a simple three-level decomposition tool, while Section 7.4 applies the properties to the decomposition choice and variable ordering used in building KDDs.

## 7.3 Three-Level Decompositions

In Chapter 2 it was explained that sum-of-products and product-of-sums representations are commonly used to describe switching functions. Both of these types of representations are called *two-level representations*, because they use two levels of operators.

A *three-level representation* is one which uses a third type of operator, such as an XOR operator. Dubrova *et. al.* have demonstrated in [57] that there exists an AND-OR-XOR representation for any Boolean function with upper bound on the number of products smaller than that for either a sum-of-products or an AND-XOR expansion. An AND-OR-XOR representation is an expression of the type

$$f(X) = (P_1 \vee P_2 \vee ... \vee P_p) \oplus (P_{p+1} \vee P_{p+2} \vee ... \vee P_m)$$

where $p \in 1 \leq p \leq m$ and $P_i$ are product terms.

AND-OR-XOR representations are suitable for implementing arithmetic functions, and can be implemented in a very simple architecture since they contain only one XOR gate [58]. The problem lies in identifying where the XOR operator should be placed. In Section 7.2 it was explained that the autocorrelation coefficients could be used to identify functions of the type $f(X) = f^*(X) \oplus g(X)$, with some limitations on $f^*(X)$ and $g(X)$. What this implies is that the autocorrelation coefficients may be used to identify AND-OR-XOR representations for functions. For example, given the function $f(X) = (x_1 \vee x_2x_3) \oplus (x_4x_5)$ the first and second order autocorrelation coefficients are shown in Figure 7.1.

As stated in Section 7.2, there are various properties of the autocorrelation coefficients that may be used in the identification of XOR logic. If any of the first order coefficients have the value $-2^n$ then the function may be decomposed into $f^*(X) \oplus x_i$. There are no coefficients of this value, so the next identifier of XOR logic is tested: the existence of three coefficients such that $x_i$ and $x_j$ take on the values 11, 01, and 10 while the remaining variables are fixed at 0 and all three of these coefficients have

| $\tau$ | $C(\tau)$ |
|---|---|
| 00001 | 0 |
| 00010 | 0 |
| 00011 | 0 |
| 00100 | 16 |
| 00110 | 0 |
| 01000 | 16 |
| 01001 | 0 |
| 01100 | 16 |
| 10000 | -16 |
| 10001 | 0 |
| 10010 | 0 |
| 10100 | -16 |
| 11000 | -16 |

**Figure 7.1.** *The autocorrelation coefficients for a function known to have a good three-level decomposition.*

the value 0. In the example shown in Figure 7.1 the only coefficients matching this are $C(00001) = 0$, $C(00010) = 0$, and $C(00011) = 0$. Therefore we have identified that the function can be decomposed into $f(X) = f^*(X) \oplus (x_4 * x_5)$ where $f^*(X)$ is independent of $x_4$ and $x_5$ and $*$ may either be the AND operator or the OR operator.

### 7.3.1 Three-Level Minimization Tools

Although this is a relatively new idea, a small number of other researchers have also developed techniques for three-level logic minimization. The work by Dubroval *et. al.* has resulted in one of these.

### Dubrova

AOXMIN-MV [59], written by E. Dubrova, is the program used for comparison to the techniques developed in this work. The pseudocode followed by AOXMIN-MV is summarized by the following steps.

- Find a cover for the on-set of the function $F$.
- Find a cover for the off-set of the function $F$.
- Check if $F$ is likely to have a compact AND-OR-XOR form as follows:
  - divide the cubes into equivalence classes of connected chains of cubes,
  - randomly partition the classes into two groups such that $F_1 \cup F_2 = F$ and $F_1 \cap F_2 = 0$, and then
  - use these to construct two sets of cubes $g_1$ and $g_2$ such that $g_1 \oplus g_2 = F$.
- The last step is to perform group migration to optimize the initial partioning of the equivalence classes of $F$.

The check step is performed for both the on-set and the off-set of the function, and if the size of $|g_1| + |g_2|$ for either is less than the smaller of the on-set and off-set then the final step of iterative optimization is performed.

This algorithm has some similarities to the process performed in this work. Both perform a preprocessing check step that provides an initial guide as to whether inclusion of XOR logic will be worthwhile. Both also use the information from this preprocessing as a guide to the decomposition into functions $g_1$ and $g_2$. However, no additional minimization or balancing is done in this work, although this could be added, and AOXMIN-MV is designed for multiple-valued multiple-output functions. The extensions to multiple-valued logic and multiple output functions are beyond the scope of this dissertation.

It should be pointed out that it is difficult to compare the processing in the check step for each algorithm since one is intended for multiple-output functions, while this work concentrates on single-output functions. Given this fact there are still similarlities in the approach. AOXMIN-MV uses the idea of chains of cubes to build equivalence classes, followed by grouping the cubes of the function according to these classes. This implies that cubes that may overlap are grouped together. Overlapping cubes imply areas of similarity in the function, since they overlap, and areas of dissimilarity, otherwise the cubes would be combined into one. Example 7.2 demonstrates this. In terms of autocorrelation coefficients, in $\{+1, -1\}$ encoding this will lead in general to smaller coefficients. The processing step in this work focuses mainly on the identification of small coefficients, thus a somewhat similar concept is being employed.

It should also be noted that AOXMIN-MV, after identifying a decomposition, performs some work to optimize the two partitions. We later refer to this as *balancing*, as the goal of this optimization is to balance the two functions into which the function is being decomposed.

| $x_3x_4 \backslash x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | |
| 01 | | | 1 | 1 |
| 11 | | | | 1 |
| 10 | | | | |

**Figure 7.2.** *A Karnaugh map demonstrating the overlapping cubes* $x_1\overline{x}_3x_4$ *and* $x_1\overline{x}_2x_4$.

### Debnath & Sasao

Debnath and Sasao present a heuristic method for minmizing AND-OR-XOR representations of $n$ variable functions in [58]. Their technique is to recursively decompose the function $F$ using Shannon's decomposition into $F_{x=1}$ and $F_{x=0}$, each requiring only $n-1$ input variables. When the number of variables that the decomposed functions require reaches 5 or fewer a table look-up for the optimum AND-OR-XOR representation is performed. This representation is then used to determine an AND-OR-XOR reprentation for the decomposed 6 variable function, and so on until an AND-OR-XOR representation for the original function $F$ is obtained.

This technique is quite different from the approach both in this work and in AOXMIN-MV, as no preliminary check is performed to identify if the use of XOR logic is advantageous. Additionally, this work appears to be aimed primarly at functions with relatively small (*i.e.* fewer than 10) numbers of inputs.

### Chattopadhyay *et. al.*

Chattopadhyay *et. al.*[60] present another three-level minimization strategy based on decomposition. Their technique uses instead the Davio decompositions. The problem they focus on is the choice of variable ordering and type of Davio expansions to choose. Their choices at each stage are affected by three factors:

- the maximization of trivial subfunctions,
- the maximization of sharing, and
- the maximization of the result of a simple cost function defined as

$$\mathrm{sim}(f) = |ones\ in\ f| - |zeros\ in\ f|$$

The Davio decompositions are first determined for all possible variables, and the above factors are used to select the variable and decomposition type resulting in the minimum number of next level gates; that is, the simplest functions and the maximum amount of sharing within the resulting implementation. After this selection, the process is repeated for the resulting decomposed functions, until only trivial (*i.e.* constant) functions remain.

Like the version presented by Sasao, no preliminary checking into whether the use of XOR logic will be advantageous is performed. However, a check was added to their tool to measure the suitability of AND-OR and AND-XOR decompositions at each level, allowing AND-OR logic to be used if it was evaluated to be better than the AND-XOR decomposition. Thus this tool could also be used in the minimization of functions that did not contain any XOR logic.

### Autosymmetries

Another technique for three-level logic minimization is based on autosymmetries [61]. The concept of autosymmetries is, as the name suggests, related to the concept of symmetries. However, a totally symmetric function may or may not be autosymmetric, and similarly an autosymmetric function may or may not be totally symmetric. The concept of autosymmetries was introduced by Luccio and Pagli in 1999 [62]. Further applications of this class of functions were detailed by Bernasconi *et. al.* in 2002 [61]. An autosymmetric function is one in which some regular, recursive structure in inherent in the outputs. This can be seen quite clearly when examining some examples of autosymmetric functions, as shown in Figure 7.3.

| $x_3x_4 \backslash x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 0 | 1 |
| 01 | 1 | 0 | 1 | 0 |
| 11 | 0 | 0 | 0 | 0 |
| 10 | 0 | 1 | 0 | 1 |

$(a)$

| $x_3x_4 \backslash x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 0 | 0 | 1 |
| 01 | 0 | 1 | 1 | 0 |
| 11 | 1 | 0 | 0 | 1 |
| 10 | 0 | 1 | 1 | 0 |

$(b)$

**Figure 7.3.** *Two functions which possess autosymmetries of degree 1 and degree 3, respectively.*

The definition of autosymmetry is as follows [61]:

**Definition 7.1** *A Boolean function $f(X)$ in $\{0,1\}^n$ is* closed *under $\alpha$, with $\alpha \in$*

$\{0,1\}^n$, *if for each* $w \in \{0,1\}^n$, $w \oplus \alpha \in f(X)$ *if and only if* $w \in f(X)$.

A vector $w \in \{0,1\}^n$ is said to be $\in f(X)$ if $w$ is a minterm for $f(X)$.

By combining under $\oplus$ $k$ linearly independent vectors $\alpha_1, \ldots, \alpha_k$ we form a subspace of $2^k$ vectors that is closed under $\oplus$. This is referred to as $L_f$, or *the linear space of* $f(X)$, and $k$ is its *dimension*.

**Definition 7.2** *A Boolean function* $f(X)$ *is* $k$-autosymmetric, *or equivalently* $f(X)$ *has* autosymmetry degree $k$, $0 \leq k \leq n$, *if its linear space* $L_f$ *has dimension* $k$.

Bernasconi *et. al.* provide an algorithm for the construction of $L_f$:

1. for all $u \in f(X)$ build the set $u \oplus f(X)$;
2. build the set $L_f \bigcap_{u \in f(X)} (u \oplus f(X))$
3. compute $k = \log_2 |L_f|$

If $k = 0$ then the function is not autosymmetric.

Classes of autosymmetric functions have very efficient sum of pseudoproduct (SPP) forms. SPPs are defined in [62]. An example function in SPP form is $(x_1 \oplus \overline{x}_2) \wedge x_3 \vee (x_1 \oplus x_3 \oplus x_4) \wedge (x_3 \oplus \overline{x}_5) \vee x_2 \wedge x_5$. According to Luccio *et. al.* the advantage of SPP forms is that they are a more general form than SOP forms, and thus a function expressed as a SPP is in the worst case the same length as the equivalent SOP, and in most cases is considerably shorter. They also have uses in three-level logic minimization, as described by Bernasconi *et. al.*

### The Autocorrelation Coefficients and Autosymmetry

The autocorrelation coefficients can be used to identify autosymmetrical functions.

**Theorem 7.1** *If a function* $f(X)$ *has an autocorrelation coefficient* $B(\tau) = B(0)$ *(*$C(\tau) = 2^n$*) then* $\tau$ *is in* $L_f$.

**Lemma 7.1** *Any function* $f(X)$ *for which one or more autocorrelation coefficient* $B(\tau) = B(0)$ *(*$C(\tau) = 2^n$*) is autosymmetrical.*

**Proof.**

First it is demonstrated that for an autosymmetric function, if a vector $\tau$ is in the linear subspace $L_f$ of the function $f(X)$ then the autocorrelation coefficient $B(\tau) = B(0) = m$ where $m$ is the number of minterms for $f(X)$.

Let $L_f = \{c_1, \ldots, c_{2^k}\}$ where $c_j \in \bigcap_{u_i \in f(X)}(u_i \oplus f(X)),\ j \in \{1, .., 2^k\}$. Additionally, $\{u_1, \ldots, u_m\}$ are the minterms for $f(X)$. If $\tau \in L_f$ then by definition

$$\tau \in \{u_1 \oplus f(X), \ldots, u_m \oplus f(X)\}$$

$u_1 \oplus f(X)$ may be rewritten as $\{u_1 \oplus u_1, \ldots, u_1 \oplus u_m\}$; similarly for $u_2, \ldots, u_m$. Then $\forall\ u_i\ \in\ f(X),\ \tau = u_i \oplus u_j,\ i, j\ \in\ \{1, \ldots, m\}$. If $u_i = u_j$ then $\tau = \mathbf{0}$.

Now let us examine the definition of the autocorrelation coefficients:

$$B(\tau) = \sum_{p=0}^{2^n-1} f(p) \times f(p \oplus \tau)$$

By definition, $f(p) = 1$ if and only if $p\ \in\ \{u_1, \ldots, u_m\}$. Then

$$\begin{aligned} B(\tau) &= \sum_{i=1}^{m} f(u_i) \times f(u_i \oplus \tau) \\ &= \sum_{i=1}^{m} f(u_i) \times f(u_j) \\ &= \sum_{i=1}^{m} 1 \times 1 \\ &= m \end{aligned}$$

Thus if $\tau \in L_f$ then $B(\tau) = m$ where $m$ is the number of positive minterms for $f(X)$.

Next it is demonstrated that if the autocorrelation coefficient $B(\tau) = m$, where $m$ is the number of minterms for the function $f(X)$, then $\tau$ is in the linear subspace $L_f$ for the function and the function is therefore autosymmetric. Again, by definition, for a given vector $p$, $f(p) = 1$ if and only if $p\ \in\ \{u_1, \ldots, u_m\}$. Then

$$\begin{aligned} B(\tau) &= \sum_{i=1}^{m} f(u_i) \times f(u_i \oplus \tau) \\ m &= \sum_{i=1}^{m} f(u_i) \times f(u_i \oplus \tau) \\ &= \sum_{i=1}^{m} 1 \times f(u_i \oplus \tau) \end{aligned}$$

therefore $u_i \oplus \tau \;\in\; \{u_1, \ldots, u_m\}$. Thus for each coefficient $B(\tau) = m$, $u_i \oplus \tau = u_j$. This can be rewritten $\tau = u_i \oplus u_j$, which leads to

$$\tau \;\in \bigcap_{u_i \;\in\; f(X)} (u_i \oplus f(X))$$

Thus the $\tau$ for coefficients with value $m$ are in $L_f$, and if there is more than one (the trivial $\tau = 0$) then the function is autosymmetric. ∎

It should be noted that the computation of $L_f$ is clearly less compute-intensive than is the computation of all $2^n$ autocorrelation coefficients. However, if the coefficients are already available, then the autosymmetry of the function can be easily determined, as shown above.

#### Summary of Tools

We have presented four tools as background for the concept of three-level minimization. Each of these tools is based on significant amounts of prior work by the authors, with in most cases, many iterations and improvements being applied to the tools in question. Two of these tools are based on applying iterative decompositions of various types, while the other two techniques require examination of the function's cubes, and grouping of the cubes to form a good decomposition. Our technique is most like this second description, in that the cubes of the function are examined to determine the autocorrelation coefficients, followed by a grouping of variables to form a decomposition. Details of our technique are given below.

### 7.3.2 Implementation of a Three-Level Decomposition Tool

#### Implementation Details

The program implemented in this work is intended as a "proof-of-concept". That is, many further enhancements could be added should one wish to make use of this three-level decomposition tool, such as the minimization of each of the resulting decompositions and extending the identification techniques to allow more than two variables in the second decomposition. However, this work is intended as an overall investigation into the uses of the autocorrelation coefficients, and thus the implementation is not intended to be an complete three-level minimization tool. We emphasize that the goal of this program is to detect the existence of a three-level decomposition; we perform no measures pertaining to the quality of the detected decomposition.

The algorithm details are as follows:

```
readcoeffs(ac_infile, coeffvector, uvector);
// identify xor logic.  NOTE:
// this is a limited version that will only id if two
// vars are in a decomp e.g. f = g xor (xi+xj)
// or f = g xor xi
id_xor(coeffvector, uvector, single_vars, double_vars);
// generate g1 and g2
generate_g1g2(dd, single_vars, double_vars, outfilename1, outfilename2);
```

This assumes that the autocorrelation coefficients have previously been generated and are required as input to the program.

The majority of the work is done in functions id_xor and generate_g1g2. id_xor does the following:

```
// first examine the first-order coeffs
// for coeffs equal to -2^n
// NOTE: we assume var ordering of 1 2 3 4 ...
// thus the corresponding first order coeff
// for variable 1 is C(1000...).
for each variable i
  if (coeffvector[twoexp(numvars-1-i)] == -2^n)
     singlevars[i] = i;
  else singlevars[i] = -1;
```

```
  // now the second order coeffs:
  // check each pair of vars to see if the corresponding
  // first and second order coeffs 000 01= 000 10 = 000 11 = zero
  initialize all entries in doublevars to -1;
  for each variable i {
    // if the var already is in the singlevar list then skip it
    if (singlevars[i] != -1) continue;
    for each variable j beginning at i+1 {
      // if the var already is in the singlevar list then skip it
      if (singlevars[i] != -1) continue;
      int uval1 = 0 | twoexp(numvars-1-j); // 000 01
      int uval2 = 0 | twoexp(numvars-1-i); // 000 10
      int uval3 = uval1 | uval2; // 000 11
      if ( (coeffvector[uval1] == 0) &&
           (coeffvector[uval2] == 0) &&
           (coeffvector[uval3] == 0) )
        doublevars[0] = i;
        doublevars[1] = j;
        goto endofloops; // don't look for any others IN THIS VERSION
    } // end for j
  } // end for i
endofloops: return;
```

generate_g1g2 creates two decision diagrams, one for the first decomposition (g1) and one for the second decomposition (g2). It then outputs them to two output files. If no XOR logic was identified by id_xor then this function informs the user.

```
  // first check if there is ANY xor logic that is usable:
  bool xorflag = false;
  check singlevars array
  check doublevars array
```

```
if (no xor logic)
  cout << "No xor logic to use\n\n";
  g1 = original dd;
  g2 = NULL;
  outputfunctions(g1, g2);

if singlevars contains xor logic
   create g1 = xi
   create g2 = dd xor g1
else, using doublevars array,
   create g1 = xi or xj
   create g2 = dd xor g1
   check g1 xor g2 = dd
   if not
     create g1 = xi and xj
     create g2 = dd xor g1
outputfunctions(g1, g2);
```

### Results

The above pseudocode was implemented and tested against the three-level minimization tool AOXMIN-MV. Details of the AOXMIN-MV tool are given in section 7.3.1. In these tests both agreed on the detection of XOR logic 74% of the time. However, the AOXMIN-MV tool only completed successfully for 244 of the single-output benchmark files, while our tool completed for all 278 of the benchmarks. Additionally, our tool required an average of approximately 5 seconds to compute the decomposition, while AOXMIN-MV, for its successful benchmarks, required an average of over one minute. It should be noted, however, that AOXMIN-MV requires more time in computation of a decomposition because the tool is also attempting to find a balanced decomposition, which our tool does not take into account. The computation of all $2^n$

autocorrelation coefficients is included in this timing figure, although future improvements could reduce this as only the first and second order coefficients are required. These results are summarized in Table 7.1.

| | successes | avg. time | num of benchmarks where XOR logic detected |
|---|---|---|---|
| AOXMIN-MV | 244 / 278 | 71.1 sec | 54 |
| 3LEVEL | 278 / 278 | 5.4 sec | 59 |

**Table 7.1.** *Results of comparing the autocorrelation-based three level-decomposition tool (3LEVEL) to AOXMIN-MV.*

It is interesting to note, however, that while our autocorrelation-based tool detected XOR logic in 59 of the benchmark files, in 32 of those the AOXMIN-MV tool did not detect XOR logic. Similarly, of the 54 benchmarks for which AOXMIN-MV detected XOR logic, our tool did *not* detect XOR logic in 27 of those. Thus for only 27 of the benchmarks did BOTH tools detect XOR logic. The immediate question is why. For the first situation, when our tool detects XOR logic while AOXMIN-MV does not, the answer is that AOXMIN-MV is attempting to find a solution for which the products are fewer than in the best two-level minimization solution. If this is not found then AOXMIN-MV does not provide a decomposition. Our tool does not take this into consideration, it simply provides the decomposition based on the theorems in Chapter 3. For the second situation, when our tool does not detect XOR logic while AOXMIN-MV does, the answer is that AOXMIN-MV takes into account more possibilities for three-level decompositions than does our tool; our tool is currently limited to only the two situations described by Theorems 3.8 and 3.9. Future work must be done to extend the tool to identify other types of decompositions.

Even given the limitations of our tool, the fact is that on 74% of the benchmarks AOXMIN-MV and our autocorrelation-based tool agree on the existence of XOR logic within the function. That these results are as good as they are is somewhat surprising, since only two types of XOR decompositions are being identified. In fact, given that this implementation is the result of the first work performed relating autocorrelation coefficients to three-level minimization while other tools represent the final of many stages of improvements, this is a very good indication that our technique is worth further investigation.

The 278 benchmarks used are described in Appendix C, and complete timing results are given in Appendix D.

## 7.4 Decomposition Type Lists for KDDs

In Chapter 2 the graph-based representations called BDDs, FDDs, and their combination KDDs were introduced. BDDs have become very popular for representing switching functions, but have exponential sizes for some classes of functions [13, 40]. The solution to this problem lies in the use of alternative decompositions, which creates a composite type of decision diagram called a KDD. KDDs have the same variable-ordering problem as do BDDs, with an added complication of deciding which type of decomposition to use at each level. The list of decompositions is usually referred to as a decomposition type list, or *dtl*. As described in Chapter 2, the choices for each decomposition may be Shannon, positive Davio, or negative Davio decompositions. Since we have shown in previous sections that the autocorrelation coefficients may be used to identify XOR logic within a function, we have further hypothesized that they also may be useful in determining a dtl for a KDD representing the function.

### 7.4.1 DTL and Ordering Tools for KDDs

The work in this dissertation is one of very few techniques that attempts to find an ordering and dtl *before* building the KDD. Most other techniques involve building the KDD and then sifting [13] or applying genetic algorithms [42]. Some of these approaches are briefly described below.

### PUMA

The PUMA-Package makes use of three sifting techniques that apply enhanced versions of the technique introduced in [13]. These techniques are referred to as Sifting, Siftlight and DTL-Sifting. Details of these are given in [63].

Our tool was compared with the DTL sifting heuristic, implemented by PUMA as DTL-Sifting. Siftlight is faster, but only performs local optimization, while Sifting only determines a variable ordering. DTL-Sifting scans all existing levels with all decomposition types before sifting the variable to another position.

It is interesting also to note that an exact algorithm called DTL_Friedman [64] is also implemented by PUMA. Initial tests to compare our results to the results of this algorithm were begun; however, for most benchmarks the algorithm took many hours to complete, so the DTL-Sifting heuristic was used instead.

### Genetic Algorithms (Drechlser *et. al.*)

Another method introduced by Drechsler *et. al.* [42] uses a genetic algorithm to determine the dtl. Once again, the process involves building an initial KDD and performing various operations on it while measuring whether the operation has improved the size or not.

### Variable Weightings

Drechsler *et. al.* also introduce another method for determining ordering and dtl in [65]. This method is more like the technique in this work in that it involves examination of the function in an alternate representation to determine the ordering and dtl before building the KDD. The authors of this work determine a number of heuristics that are intended for application on differing types of circuits; for example, KDDs for tree-like circuits are treated differently from KDDs for two-level circuits, or KDDs for multi-level circuits.

Since our work uses a two-level representation of the function to compute the autocorrelation coefficients, we contrast the two-level heuristics of Drechsler *et. al.* with our technique. In the two-level heuristics, a weight value for each variable is computed. The weight value is determined by comparing $plit_i$, the number of positive occurrences of the literal $x_i$, with $nlit_i$, the number of negative occurrences of the literal $x_i$ ($\overline{x}_i$), and then comparing the resulting cost with an arbitrarily chosen parameter $k \in [0, 0.5]$. The weight value is defined as

$$var\_weight_i = \begin{cases} plit_i + nlit_i & : nlit_i = plit_i = 0 \vee k = 0 \\ \frac{plit_i}{plit_i + nlit_i} & : \frac{plit_i}{plit_i + nlit_i} \leq k \\ \frac{nlit_i}{plit_i + nlit_i} & : \frac{nlit_i}{plit_i + nlit_i} \leq k \\ plit_i + nlit_i & : else \end{cases}$$

Depending on the value of the variable weight the decomposition type is chosen – the first and last cases indicate that the decomposition type should be Shannon. In these cases the difference between the number of positive literals and negative literals is small. In the second case a negative Davio node is chosen. This is the case if there are many more negative literals than positive ones. In the third case, when there are many more positive literals, a positive Davio node is chosen.

This technique is most like the autocorrelation-based technique since it also examines the function prior to building. However, it appears that the choice of decomposition types is almost the opposite of that made by the autocorrelation-based technique. In our technique a variable with a first order coefficient of $-2^n$, indicating many areas of the function that are dissimilar, leads to a Davio node, while in the Drechsler technique a variable with similar numbers of positive and negative literals – *i.e.* dissimilarity in the function – leads to a choice of a Shannon node. One explanation for this is that the *ordering* of the dtl may be as important as the actual dtl itself.

### 7.4.2 Implementation of a KDD Ordering and Decomposition Tool

#### Implementation Details

A simple program was written to implement an algorithm that uses the autocorrelation coefficients to determine a variable ordering and dtl for building kdds. The algorithm used for determining a variable ordering was taken from [9]. The algorithm for determining a dtl is based on the autocorrelation properties used for identification of XOR logic. Pseudocode is given below. The ordering and dtl was used to build KDDs using the PUMA [63] decision diagram package. For each benchmark function a dtl and ordering was determined, and then the KDD for the function was built. The size of the KDD and the time required to compute the autocorrelation coefficients and determine the ordering and dtl were then compared against the size and time required by a heuristic making use of sifting that is provided in the PUMA package. These heuristics are based on the concept introduced in [13].

The ordering and dtl tool performs two passes examining the autocorrelation coefficients:

- one pass to generate a variable ordering, and
- one pass to generate a dtl.

Some information about the ordering is used in generating the dtl. The ordering is generated by creating two lists; one storing the first order coefficient values, and the second storing the corresponding variables. A sort is then performed to order the values from smallest to largest, with all swap operations being performed on both lists. The resulting ordering of variables is returned for use in building the KDD.

The dtl generation function is more complex. First the first order autocorrelation coefficients are examined. If any of these match the required pattern for XOR logic, the variable level is assigned a negative Davio decomposition type. The second step is to examine the second order autocorrelation coefficients. If any of these match the pattern for XOR logic, then one of them is assigned a positive Davio decomposition. Choices of positive or negative decompositions were made through experimentation. Pseudocode for this function is given below.

```
generate_dtl(coeffvector, ordering, dtl)
  // first examine the first-order coeffs:
  for each variable i
    if (coeffvector[twoexp(numvars-1-i)] = negtwoexpn)
      dtl[i] = negDavioType;
    else dtl[i] = shannonType;

  // now the examine the second order coeffs:
  // check each pair of vars to see if the corresponding
  // first and second order coeffs 000 01= 000 10 = 000 11 = zero

  for each variable i
    // if the var already has a davio type then skip it
    if (dtl[i] != shannonType) continue;

    for each variable j starting at i+1
      // if the var already has a davio type then skip it
      if (dtl[i] != shannonType) continue;

      int uval1 = 0 | twoexp(numvars-1-j); // 000 01
      int uval2 = 0 | twoexp(numvars-1-i); // 000 10
      int uval3 = uval1 | uval2; // 000 11

      if (coeffvector[uval1] = coeffvector[uval2] = coeffvector[uval3] = 0)
        // if one of them is first in the ordering, then
        // make the other the davio node, but just one
        // becomes a davio node..
```

```
if (lab[0] != i) dtl[j] = posDavioType;
else dtl[j] = posDavioType;
```

### Results

We find that for only 4 benchmarks does the autocorrelation-based ordering and decomposition result in a smaller KDD than that determined by the sifting heuristic. However, the average number of nodes for the two techniques are very close, and if a one-node size difference is permitted then the two methods perform equally well for 173 (63%) of the benchmarks. A summary of the results is given in Table 7.2 while the complete results are given in Appendix D. In both sets of tables the comparison tool is referred to as *DTL_SIFT* while our autocorrelation-based method is referred to as the *AC method*.

| | successes | avg. time | avg. nodes |
|---|---|---|---|
| DTL_SIFT | 276 / 278 | 0.09 sec | 14.0 |
| AC method | 276 / 278 | 0.43 sec | 16.5 |

**Table 7.2.** *Summary of results comparing the DTL_SIFT heuristics implemented in the PUMA KDD package to our autocorrelation-based dtl tool.*

As indicated in Table 7.2, for both methods the KDD-builder failed to build a KDD for 2 benchmarks. This turned out to be caused by the fact that both of these benchmarks have no minterms.

While these results indicate that the sifting method performs better than our method, the difference in the performance is almost negligible. This is interesting, particularly when the following is considered:

- The autocorrelation-based ordering heuristic used was the simplest of those introduced in [9]; work in this area would likely lead to a further reduction in KDD sizes.

- The autocorrelation-based dtl heuristic examines the autocorrelation coefficients for only two relatively limited variants including XOR logic. Extension of this process would likely identify other situations in which a Davio decomposition would result in smaller or fewer subtrees than a Shannon decomposition.
- Only experimental fine-tuning was performed with respect to whether a positive or a negative Davio decomposition would be the best choice at each level. Further analysis is required in order to determine if the autocorrelation coefficients provide information that may be used in this area.

In fact, analysis of some of the benchmarks for which our tool gave particularly bad performance shows that in most cases, Shannon decompositions were chosen for all of the variables. This implies that our algorithm found no matches for the XOR-logic patterns in the autocorrelation coefficients. DTL_SIFT, on the other hand, did choose non-Shannon decompositions for these benchmarks. This leads to the conclusion that there are other factors indicating a Davio decomposition is a good choice which are not being detected by our algorithm. Again, we are aware that our tool is only currently designed to detect two situtations in which the inclusion of XOR-based decompositions may lead to better KDDs. Clearly this is an area where future work in extending the algorithm may lead to exceptional improvements. It should also be noted that the timing for the autocorrelation-based tool includes computation of all $2^n$ coefficients. This could be considerably reduced if only the required first and second-order coefficients were computed.

## 7.5 Conclusion

This Chapter presents two applications of the autocorrelation coefficients, one in identifying three-level decompositions and one in the determination of decomposition types for KDD nodes. Based on the theorems presented in Chapter 3 a proof-of-concept tool for each application has been implemented and tested against known tools. The results for each are very good. We found that our tools perform as well as the comparison tools 74% of the time for the three-level decomposition, and 63% of the time for the KDD dtl and ordering determination, and both perform very quickly. Given the simplicity of the algorithms, these results are very exciting, and certainly indicate the value of further work in these areas.

# Chapter 8

# Conclusion

This chapter summarizes the research contributions of this dissertation, and discusses several potential directions for future research related to this work.

## Contributions

This dissertation details an investigation into the uses of the autocorrelation coefficients in digital logic applications, primarily logic synthesis. Before performing an in-depth investigation of this sort, we deemed it necessary to have a fast and efficient technique for the computation of these coefficients. Additionally, since the basic properties of the autocorrelation coefficients had not been documented, this is also an area addressed by this dissertation. The main areas of application that are considered for the autocorrelation coefficients are the classification of switching functions, three-level decomposition, symmetry-testing, and DD construction. The primary contributions of this work are as follows:

- theorems relating the autocorrelation coefficients to underlying properties such as sparcity, degenerateness, and the existence of XOR logic;
- theorems relating the coefficients to symmetries in the original switching function, and the identification of a new type of symmetry called *anti-symmetries*;
- two fast DD-based techniques for the computation of single autocorrelation coefficients;

- an autocorrelation-based classification technique that encompasses the spectral classes, as well as the determination of a canonical representative for each class;
- an algorithm for the detection of XOR logic leading to a three-level decomposition of a switching function; and
- an algorithm for the detection of XOR logic leading to a variable ordering and dtl for the construction of a KDD representing the function.

## Future Work

There are many areas in which the above contributions may be applied or extended. The first, and main concern for researchers continuing this work must certainly be to extend the above results and applications to incompletely specified and multiple-output functions. Since most switching functions for practicial applications have multiple outputs, extending the properties and classification techniques is likely to be the first area for future consideration. Also, as briefly mentioned in Chapter 5, it is feasible to extend the BDD techniques introduced in this chapter to the computation of multiple-output functions.

Similarly, there is potential for these results to be applied to multiple-valued logic functions. This is an area of increasing popularity, and the extensibility of this work into the multiple-valued case should be investigated.

Additional research should also be performed with regards to the three-level decomposition and KDD dtl-determination tools. The present algorithms certainly perform well, agreeing with existing tools for 74% and 63% of the tested benchmarks respectively. However, these tools currently detect XOR logic involving either one or two input variables. There are many other cases that may exist, and this is certainly is worth investigation. As the tools stand the results are very good; it is also possible that some further optimization of how the variables are chosen for the decompositions or dtl would provide even better results. In particular, future directions for the three-level decomposition tool involve investigating techniques for balancing the decompositions, such as recursively re-applying our technique to the larger of the functions or identification of more generalized patterns within the autocorrelation coefficients. Work on the KDD dtl tool must also refine the choices of node decompositions, and we envision this work incorporating symmetry and anti-symmetry information, as well as making use of more generalized Theorems for the identification of XOR decompositions.

Finally, there are many interesting potential uses for the classification technique presented in Chapter 6. Some of these uses include the use of reconfigurable logic to implement a canonical representative of a class, and the identification of certain classes for which a KDD/FDD/BDD may provide the most efficient representation. These are areas for which applications are discussed, but have not been tested. It would be very interesting to develop and implement algorithms for each of these concepts.

# Bibliography


[1] J. Wakerly, *Digital Design Principles and Practice*, Prentice Hall, 1990.

[2] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Comp.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[3] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," *ACM Computing Surveys*, vol. 24, no. 3, Sept. 1992.

[4] R. Drechsler and D. Sieling, "Binary Decision Diagrams in Theory and Practice," *Int. Journal on Software Tools for Technology Transfer (STTT)*, pp. 112–136, May 2001.

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

[6] S. Aborhey, "Autocorrelation Testing of Combinational Circuits," *Computers and Digital Techniques, IEE Proceedings E*, vol. 136, no. 1, pp. 57–61, Jan. 1989.

[7] R. Tomczuk, *Autocorrelation and Decomposition Methods in Combinational Logic Design*, Ph.D. thesis, University of Victoria, 1996.

[8] ed. T. Sasao, *Logic Synthesis and Optimization*, Kluwer Academics, 1993, Chapter 10, Efficient Spectral Techniques for Logic Synthesis by D. Varma and E. A. Trachtenberg.

[9] J. E. Crow[1], "Variable Ordering for ROBDD-Based FPGA Logic Synthesis," M.S. thesis, University of Victoria, 1995.

[10] M. Karpovsky, R. Stankovic, and J. Astola, "Construction of Linearly Transformed Binary Decision Diagrams by Autocorrelation Functions," in *Proceedings of the International TICSP Workshop on Spectral Methods and Multirate Signal Processing, SMMSP'2001, Pula, Croatia, June 16-18*, 2001.

[11] S. Hurst, *The Logical Processing of Digital Signals*, Crane Russak, 1978.

[12] C. Edwards, "The Application of the Rademacher-Walsh Transform to Boolean Function Classification and Threshold Logic Synthesis," *IEEE Trans. on Comp.*, pp. 48–62, Jan. 1975.


[1] The author's last name has since been changed to Rice.

[13] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proceedings of ICCAD*, 1993.

[14] G. Boole, "The Calculus of Logic," *The Cambridge and Dublin Mathematical Journal*, vol. 3, 1848.

[15] C. E. Shannon, "A Symbolic Analysis of Relay and Switching Circuits," *Trans. AIEE*, vol. 57, pp. 713–723, 1938, Dissertation, Elec. Engrg. Dept., Mass. Inst. Tech., Cambridge, 69, 1940.

[16] F. Mailhot and G. De Micheli, "Algorithms for Technology Mapping Based on Binary Decision Diagrams and on Boolean Operations," *IEEE Trans. on CAD*, vol. 12, no. 3, pp. 599–620, May 1993.

[17] R. Brayton, G. Hachtel, and A. Sangiovanni-Vincentelli, "Multilevel Logic Synthesis," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 264–300, Feb. 1990.

[18] N. Woo, "A Heuristic Method for FPGA Technology Mapping Based on the Edge Visibility," in *Proceedings of DAC*, June 1991, pp. 248–251, Paper 15.5.

[19] A. Sangiovanni-Vincentelli, A. El Gamal, and J. Rose, "Synthesis Methods for Field Programmable Gate Arrays," *Proceedings of the IEEE*, vol. 81, no. 7, pp. 1057–1083, July 1993.

[20] M. Karnaugh, "The Map Method for Synthesis of Combinational Logic Circuits," *Trans. AIEE. pt I*, vol. 72, no. 9, pp. 593–599, November 1953.

[21] K. Karplus, "Using if-then-else DAGs for Multi-Level Logic Minimization," Tech. Rep. UCSC-CRL-88-29, University of California Santa Cruz, Nov. 1988, ftp'd from ftp.cse.ucsc.edu, 20 pages.

[22] C. Y. Lee, "Representation of Switching Circuits by Binary Decision Diagrams," *Bell System Technical Journal*, pp. 958–999, 1959.

[23] S. B. Akers, "On a Theory of Boolean Functions," *Journal of the Society of Industrial and Applied Mathematics*, pp. 487–498, Dec. 1959.

[24] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Comp.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[25] C. Meinel and T. Theobald, *Algorithms and Data Structures in VLSI Design*, Springer-Verlag, 1998.

[26] M. Karpovsky, *Finite Orthogonal Series in the Design of Digital Devices*, John Wiley & Sons, 1976.

[27] M. Thornton and V. S. S. Nair, "Efficient Calculation of Spectral Coefficients and their Application," *Trans. on CAD*, pp. 1328–1341, Nov. 1995.

[28] D. M. Miler, "Graph Algorithms for the Manipulation of Boolean Functions and their Spectra," *Congressus Numerantium*, pp. 177–199, 1987.

[29] C. Edwards, "The Generalised Dyadic Differentiator and its Application to 2-valued Functions Defined on an n-space," *Computers and Digital Techniques*, vol. 1, no. 4, pp. 137–142, Oct 1978.

[30] R. S. Stanković, "Some Remarks on Gibbs Derivatives on Finite Dyadic Groups," in *Theory and Applications of Gibbs Derivatives, Proceedings of the First International Workshop on Gibbs Derivatives*, 1989, pp. 159–180.

[31] C. Edwards and S. L. Hurst, "A Digital Synthesis Procedure Under Function Symmetries and Mapping Methods," *IEEE Trans. on Comp.*, Nov. 1978.

[32] R. Tomczuk, *Autocorrelation and Decomposition Methods in Combinational Logic Design*, Ph.D. thesis, University of Victoria, 1996.

[33] C.-C. Tsai and M. Marek-Sadowska, "Boolean Functions Classification via Fixed Polarity Reed-Muller Forms," *IEEE Trans. on Comp.*, 1997.

[34] B. Kim and D. L. Dietmeyer, "Multilevel Logic Synthesis of Symmetric Switching Functions," *IEEE Trans. on CAD*, pp. 436–446, Apr. 1991.

[35] C.-C. Tsai and M. Marek-Sadowska, "Generalized Reed-Muller Forms as a Tool to Detect Symmetries," *IEEE Trans. on Comp.*, pp. 33–40, Jan. 1996.

[36] V. N. Kravets and K. A. Sakallah, "Constructive Library-Aware Synthesis Using Symmetries," in *Proceedings of DATE*, 2000.

[37] L. Heinrich-Litan and P. Molitor, "Least Upper Bounds for the Size of OBDDs Using Symmetry Properties," *IEEE Trans. on Comp.*, pp. 360–368, Apr. 2000.

[38] S. Panda, F. Somenzi, and B. Plessier, "Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams," in *Proceedings of ICCAD*, 1994.

[39] C. Scholl, D. Möller, P. Molitor, and R. Drechsler, "BDD Minimization using Symmetries," *IEEE Trans. on CAD*, pp. 81–99, Feb. 1999.

[40] Randal E. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication," *IEEE Trans. on Comp.*, vol. 40, no. 2, pp. 205–213, Feb. 1991.

[41] C. Scholl, S. Melchior, G. Hotz, and P. Molitor, "Minimizing ROBDD Sizes of Incompletely Specified Boolean Functions by Exploiting Strong Symmetries," in *Proceedings of ED&TC*, 1997, pp. 229–234.

[42] R. Drechsler and B. Becker, "Sympathy: Fast Exact Minimization of Fixed Polarity Reed-Muller Expressions for Symmetric Functions," in *ED&TC*, 1995.

[43] J. E. Rice, J. C. Muzio, and M. Serra, "The Use of Autocorrelation Coefficients for Variable Ordering for ROBDDs," in *Proceedings of the 4th International Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, 1999.

[44] F. Somenzi, "CUDD: Colorado University Decision Diagram Package," version 2.3.0, Department of Electrical and Computer Engineering, University of Colorado at Boulder, *Fabio@Colorado.EDU*.

[45] J. E. Rice and J. C. Muzio, "Methods for Calculating Autocorrelation Coefficients," in *Proceedings of the 4th International Workshop on Boolean Problems, (IWSBP2000)*, 2000, pp. 69–76.

[46] B. Falkowski, I. Schaefer, and M. A. Perkowski, "Calculation of the Rademacher-Walsh Spectrum from a Reduced Representation of Boolean Functions," in *Proceedings of DAC*, 1992.

[47] S. Devadas, D. Varma, and E. A. Trachtenberg, "Design Automation Tools for Efficient Implementation of Logic Functions by Decomposition," *IEEE Trans. on CAD*, vol. 8, no. 8, Aug. 1989.

[48] B. Falkowski, I. Schaefer, and M. A. Perkowski, "A Fast Computer Algorithm for the Generation of Disjoint Cubes for Completely and Incompletely Specified Boolean Functions," in *Proceedings of the 33rd IEEE Midwest Symposium on Circuits and Systems*, 1990.

[49] D. Wessels, "Randomised Techniques to Efficiently Approximate Spectral Coefficients and Autocorrelation Coefficients," Tech. Rep., James Cook University of North Queensland, 1996.

[50] D. Wessels, "Efficient Approximation of Spectral and Autocorrelation Coefficients," in *Proceedings of the IEEE Region Ten Conference on Digital Signal Processing Applications (TENCON)*, 1996.

[51] R. Karp, M. Luby, and N. Madras, "Monte Carlo Approximation Algorithms for Enumeration Problems," *Journal of Algorithms*, pp. 429–228, 1989.

[52] Robert Lisanke, "Logic Synthesis and Optimization Benchmarks User Guide Version 2.0," 1988, a report to provide documentation for benchmark examples used in conjunction with the 1989 MCNC International Workshop on Logic Synthesis.

[53] S. Brown, R. Francis, J. Rose, and Z. Vranesic, *Field Programmable Gate Arrays*, Kluwer Academic Publishers, 1992.

[54] E. L. Horta, J. W. Lockwood, and S. T. Kofuji, "Using PARBIT to Implement Partial Run-Time Reconfigurable Systems," in *Proceedings of Field-*

*Programmable Logic and Applications (FPL) 2002*. 2002, pp. 182–191, Springer Lecture Notes in Computer Science (LNCS).

[55] E. Dubrova and T. Bengtsson, "A Sufficient Condition for Detection of XOR-Type Logic," in *Proceedings of of NORCHIP*, 2001, pp. 271–279.

[56] B. Becker, R. Drechsler, and M. Theobald, "OKFDDs versus OBDDs and OFDDs," in *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, 1995.

[57] E. V. Dubrova, D. M. Miller, and J. C. Muzio, "Upper Bound on Number of Products in AND-OR-XOR Expansion of Logic Functions," *IEE Electron. Lett.*, vol. 31, pp. 541–542, 1995.

[58] D. Debnath and T. Sasao, " A Heuristic Algorithm to Design AND-OR-EXOR Three-Level Networks," in *Proceedings of ASP-DAC*, 1998, pp. 69 –74.

[59] E. Dubrova, "AOXMIN-MV: A Heuristic Algorithm for AND-OR-XOR Minimization," in *Proceedings of the 4th International Workshop on Applications of Reed-Muller Expansions in Circuit Design (RM99)*, 1999, pp. 37–53.

[60] S. Chattopadhyay, S. Roy, and P. P. Chaudhuri, " KGPMIN: An Efficient Multilevel Multioutput AND-OR-XOR Minimizer," *IEEE Trans. on CAD*, vol. 16, no. 3, pp. 257–265, March 1997.

[61] A. Bernasconi, V. Ciriani, F. Luccio, and L. Pagli, "Fast Three-Level Logic Minimization Based on Autosymmetry," in *Proceedings of DAC*, 2002.

[62] F. Luccio and L. Pagli, "On a New Boolean Function With Applications," *IEEE Trans. on Comp.*, , no. 3, pp. 296–310, March 1999.

[63] Andreas Hett, "PUMA," 1995, decision diagram software, see http://ira.informatik.uni-freiburg.de/software/puma/pumamain.html.

[64] S. Friedman and K. Supowit, "Finding the Optimal Variable Ordering for Binary Decision Diagrams," in *Proceedings of DAC*, 1987, pp. 348–355.

[65] R. Drechsler, B. Becker, and A. Jahnke, "On Variable Ordering and Decomposition Type Choice in OKFDDs," *Trans. on Computers*, vol. 47, no. 12, pp. 1398–1403, Dec. 1998.

[66] R. Brayton, G. Hactel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

# Appendix A

# List of Notation and Symbols Used

## General Notation

A Boolean, or switching function is a function $f(X)$ in which

$$f(X) \in \{0, 1\}$$

$$X = \{x_1, ..., x_{n-1}, x_n\}$$

$$x_i \in \{0, 1\} \ \forall \ i \in 1...n$$

In any truth table defining a switching function $f(X)$ the highest order bit (leftmost) is labeled $x_n$ while the lowest order bit (rightmost) is labeled $x_1$ as shown in Figure A.1.

The Boolean operators AND, OR, negation (complementation) and exclusive-OR are represented by the symbols in Table A.1.

## Output Encoding

The output vector of a Boolean function $f(X)$ is referred to as $Z$ if $\{0, 1\}$ encoding is used, and as $Y$ if $\{+1, -1\}$ encoding is used. $z_i$ and $y_i$ refer to individual elements of the vectors $Z$ and $Y$ respectively.

| $x_3$ | $x_2$ | $x_1$ | $f(X)$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

**Figure A.1.** *A truth table demonstrating how the input bits are labeled.*

| Operator | Symbol |
|---|---|
| AND | $\wedge$ |
| OR | $\vee$ |
| exclusive-OR | $\oplus$ |
| negation of variable $x$ | $\overline{x}$ |

**Table A.1.** *The symbols used to represent the most common Boolean operators.*

# Notation for Spectral Coefficients

The spectral coefficient vectors are labeled $R$ and $S$ for $\{0, 1\}$ and $\{+1, -1\}$ encoding respectively as shown in Figure A.2.

$s_i$ and $r_i$ refer to individual elements of the $S$ and $R$ vectors respectively.

$$
\begin{array}{ll}
\{0,1\} & T^n \cdot Z = R \\
\{+1,-1\} & T^n \cdot Y = S
\end{array}
$$

**Figure A.2.** *The spectral transforms for computing $R$ and $S$.*

# Notation for Autocorrelation Coefficients

The autocorrelation function is defined as

$$B^{f\,f}(\tau) = \sum_{v=1}^{2^n-1} f(v) \cdot f(v \oplus \tau)$$

where

$$v = \sum_{i=1}^{n} v_i \cdot 2^{i-1}$$
$$\tau = \sum_{i=1}^{n} \tau_i \cdot 2^{i-1}$$

Alternatively, $\tau$ may be replaced with $u$.

If $\{+1,-1\}$ encoding is used then the resulting coefficient is labeled $C^{f\,f}(\tau)$.

In general, the superscript $f\ f$ is omitted when referring to the autocorrelation function. For the remainder of this dissertation, $B$ ($C$) is used to refer to the entire vector of autocorrelation coefficients, and $B(\tau)$ ($C(\tau)$) is used to refer to each entry in this vector. Figure A.3 illustrates three alternative labelings for $B(\tau)$.

$$
\begin{bmatrix} B(0) \\ B(0) \\ B(2) \\ B(3) \\ B(4) \\ B(5) \\ B(6) \\ B(7) \end{bmatrix}
=
\begin{bmatrix} B(000) \\ B(001) \\ B(010) \\ B(011) \\ B(100) \\ B(101) \\ B(110) \\ B(111) \end{bmatrix}
=
\begin{bmatrix} B(0) \\ B(x_1) \\ B(x_2) \\ B(x_1x_2) \\ B(x_3) \\ B(x_1x_3) \\ B(x_2x_3) \\ B(x_1x_2x_3) \end{bmatrix}
$$

**Figure A.3.** *Alternative labelings for the of the autocorrelation coefficients (assuming $n = 3$).*

# Appendix B

# Glossary

## B.1 Acronyms

**ASIC** - Application Specific Integrated Circuit

**BDD** - Binary Decision Diagram

**DAG** - Directed Acyclic Graph

**dtl** - decomposition type list

**FDD** - Functional Decision Diagram

**FPGA** - Field Programmable Gate Array

**KDD** - Kronecker Desision Diagram

**LUT** - Look-up Table

**NP** - Non-deterministic Polynomial

**NPN** - Negation (inputs), Permutation, Negation (outputs)

**PAL** - Programmable Array Logic

**PLA** - Programmable Logic Array

**PLD** - Programmable Logic Device

**PML** - Programmable Macro Logic

**POS** - Product of Sums

**ROBDD** - Reduced Ordered Binary Decision Diagram

**RTL** - Register Transfer Language

**SOP** - Sum of Products

**VHDL** - VHSIC Hardware Description Language

**VHSIC** - Very High Speed Integrated Circuits

**XOR** - exclusive-or

## B.2 Definitions

**analysis** - the process of extracting a formal description of a function from a diagram of the circuit implementation

**BDD** - Binary Decision Diagram. A binary directed acyclic graph with two leaves *TRUE* and *FALSE*, in which each non-leaf node is labeled with a variable and has two out-edges, one pointing to the subgraph that is evaluated if the node label evaluates to *TRUE* and the other pointing to the subgraph that is evaluated if the node label evaluates to *FALSE*.

**Chow parameters** - $n+1$ parameters defined as

$$CH(X) \triangleq Ch(x_1), Ch(x_2), ..., Ch(x_n); Ch(x_0)$$
$$where$$
$$Ch(x_i) = \sum occurrence\ \ of\ \ x_i\ \ over\ \ all\ \ true\ \ minterms;\ \ i = 1\ \ to\ \ n$$
$$Ch(x_0) = total\ \ number\ \ of\ \ true\ \ minterms$$

**completely specified function** - a Boolean function in which the output is defined for all $2^n$ possible input combinations

**complexity measure** - the complexity measure $C(f)$ is defined as

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

**cover** - a cover of an incompletely specified function $F$ (*i.e.* a function with a don't care set) is a completely specified function $f$ such that all minterms in the on-set of $f$ are also in $F$ and no minterm in $F$ is in the off-set of $f$

**degenerate function** - a function of $n$ variables wich does not depend upon all $n$ inputs to determe the function output(s)

**FPGA** - Field Programmable Gate Array. A FPGA is a device in which the final logic structure can be directly configured by the end user, without the use of an integraged circuit fabrication facility

**literal** - a variable $x_i$ or its complement $\overline{x_i}$

**linearly separable** - a term used to describe a function for which, when its minterms are considered as $2^n$ equispaced nodes in $n$-dimensional space, there exists a plane that separates all the true ($f(X) = 1$) nodes from all the false nodes ($f(X) = 0$); also known as a threshold function

**LUT** - Look-up Table. A memory table that can look up any of $2^{2^n}$ possible functions, where $n$ is the number of inputs

**maxterm** - a sum term in which each of the $n$ variables $x_i$ appears exactly once as either $x_i$ or $\overline{x_i}$

**maxterm expansion** - a canonical product-of-sums form formed by multiplying the maxterms for which $f(X) = 0$ and in which no identical maxterms appear more than once

**minterm** - a product term in which each of the $n$ variables of a switching function appears exactly once in either its true or complemented form

**PLA** - A Programmable Logic Array. A PLA is a circuit consisting of a grid that implements a two-level AND/OR circuit

**product** - the result of an AND operation

**product term** - a literal or a product of literals

**product-of-sums** - a Boolean function representation consisting of a product of sum terms

**ROBDD** - a Reduced, Ordered Binary Decision Diagram (see BDD). A BDD is a *reduced* BDD if it contains no vertex whose left subgraph is equal to its right subgraph, nor does it contain distinct vertices $v$ and $v'$ such that the subgraph rooted by $v$ and $v'$ are isomorphic, and a BDD is an *ordered* BDD if on every path from the root node to an output, the variables are encountered in the specified order.

**self-dual function** - the self dual of a function is written $f^D(X)$ and is defined as $f^D(X) \triangleq \overline{f(\overline{X})}$

**Shannon's expansion** $f(X) = x_i f_{x_i=1} \oplus \overline{x}_i f_{x_i=0}$

**sum** - the result of an OR operation

**sum term** - a literal or a sum of literals

**sum-of-minterms** - a canonical sum-of-products representation of a Boolean function, consisting of a sum of minterms for which $f(X) = 1$ and in which no two identical minterms appear

**sum-of-products** - a Boolean function representation consisting of a sum of product terms

**symmetry** - a function $f(X)$ is said to be symmetrical, or to have symmetry, in a subset of its variables if $f$ remains unchanged for any permutation of those variables

**synthesis** - the process of taking a formal description of a function and developing a diagram representing a circuit implementation

**trivial function** - a function for which the output is constant (that is, $f(X) = 0$ or $f(X) = 1$ for Boolean functions)

**threshold function** - a function for which, when its minterms are considered as $2^n$ equispaced nodes in $n$-dimensional space, there exists a plane that separates all the true ($f(X) = 1$) nodes from all the false nodes ($f(X) = 0$); also known as a linearly separable function

# Appendix C

# Benchmarks

The benchmarks used in this work are from various sources, including the MCNC 1989 series of benchmarks [52] and those distributed with the ESPRESSO logic minimization tool [66].

Because this work is limited to single-output functions, each of the benchmarks have been separated into separate files for each of their outputs. The following tables list all of the benchmarks as a whole, rather than as the separate filenames that were assigned to each output.

| filename | inputs | outputs | products | filename | inputs | outputs | products |
|---|---|---|---|---|---|---|---|
| C17 | 5 | 2 | 7 | pm1 | 4 | 10 | 40 |
| ex3 | 5 | 1 | 4 | radd | 8 | 5 | 75 |
| misex1 | 8 | 7 | 32 | term1 | 34 | 10 | 257 |
| o64 | 130 | 1 | 65 | tial | al | 8 | 586 |
| sao2 | 10 | 4 | 58 | ttt2 | 24 | 21 | 149 |
| t481 | 16 | 1 | 481 | unreg | 36 | 16 | 48 |
| z4 | 7 | 4 | 59 | vda | 17 | 39 | 793 |
| add6 | 12 | 7 | 355 | vg2 | 25 | 8 | 110 |
| b1 | 45 | 45 | 1224 | x2dn | 82 | 56 | 105 |
| cm42a | 4 | 10 | 40 | dk27 | 9 | 9 | 10 |
| apex1 | 45 | 45 | 206 | C7552 | 5 | 16 | 16 |
| apex7 | 49 | 37 | 517 | alu1 | 12 | 8 | 19 |
| apla | 10 | 12 | 26 | alu2 | 10 | 6 | 261 |
| b9 | 41 | 21 | 138 | alu4 | 14 | 8 | 1799 |
| clip | 9 | 5 | 167 | bca | 26 | 46 | 180 |
| cm152a | 11 | 1 | 8 | bcb | 26 | 39 | 155 |
| cm82a | 5 | 3 | 23 | bcd | 26 | 38 | 117 |
| con1 | 7 | 2 | 9 | bw | 5 | 28 | 87 |
| in0 | n0 | 11 | 107 | cht | 47 | 36 | 120 |
| dc2 | 8 | 7 | 40 | cu | 14 | 11 | 22 |
| bc0 | 26 | 11 | 179 | decod | 5 | 16 | 16 |
| cmb | 16 | 4 | 54 | dk17 | 10 | 11 | 18 |
| cm85a | 11 | 3 | 48 | dk48 | 15 | 17 | 22 |
| co14 | 14 | 1 | 47 | mux | 21 | 1 | 36 |
| dc1 | 4 | 7 | 9 | duke2 | 22 | 29 | 87 |
| dist | st | 5 | 121 | f51m | 14 | 8 | 1799 |
| f2 | 4 | 4 | 12 | in2 | n2 | 10 | 137 |
| frg2 | 143 | 139 | 4377 | in3 | n3 | 29 | 74 |
| in1 | 16 | 17 | 104 | in4 | n4 | 20 | 212 |
| ldd | 47 | 36 | 120 | in5 | n5 | 14 | 62 |
| life | 9 | 1 | 512 | in6 | n6 | 23 | 54 |
| max46 | 9 | 1 | 46 | in7 | n7 | 10 | 54 |
| pcler8 | 16 | 5 | 45 | jbp | 36 | 57 | 122 |

| filename | inputs | outputs | products | filename | inputs | outputs | products |
|---|---|---|---|---|---|---|---|
| k2 | 45 | 45 | 1224 | 5xp1 | 7 | 10 | 75 |
| parity | 16 | 1 | 32768 | 9symml | 9 | 1 | 87 |
| mlp4 | 8 | 8 | 128 | C5315 | 49 | 37 | 517 |
| rckl | 32 | 7 | 32 | adr4 | 8 | 5 | 75 |
| rd53 | 5 | 3 | 32 | alu3 | 10 | 8 | 66 |
| root | 8 | 5 | 57 | apex2 | 39 | 3 | 1035 |
| sct | 19 | 15 | 74 | bcc | 26 | 45 | 137 |
| sym10 | 10 | 1 | 837 | cc | 21 | 20 | 52 |
| x1 | 51 | 35 | 324 | chkn | 29 | 7 | 140 |
| too_large | 38 | 3 | 1075 | cm151a | 19 | 9 | 45 |
| x2 | 10 | 7 | 35 | cm162a | 27 | 17 | 61 |
| x7dn | 66 | 15 | 539 | cm163a | 16 | 13 | 37 |
| x9dn | 27 | 7 | 120 | ex2 | 5 | 1 | 7 |
| z4ml | 7 | 4 | 59 | example2 | 10 | 6 | 261 |
| apex3 | 54 | 50 | 280 | lal | 21 | 20 | 52 |
| apex4 | 9 | 19 | 438 | frg1 | 28 | 3 | 119 |
| apex5 | 117 | 88 | 1227 | gary | 15 | 11 | 107 |
| apex6 | 143 | 139 | 4377 | log8mod | 8 | 5 | 47 |
| cm150a | 21 | 1 | 17 | majority | 5 | 1 | 5 |
| count | 35 | 16 | 184 | misex2 | 25 | 18 | 29 |
| e64 | 65 | 65 | 65 | mish | sh | 43 | 82 |
| ex1 | 5 | 1 | 16 | pcle | 18 | 9 | |
| misex3 | 14 | 14 | 1848 | rd73 | 7 | 3 | 141 |
| misex3c | 14 | 14 | 305 | risc | sc | 31 | 28 |
| misg | 56 | 23 | 69 | ryy6 | 16 | 1 | 112 |
| seq | 41 | 35 | 1459 | tcon | 17 | 16 | 32 |
| rd84 | 8 | 4 | 256 | x1dn | 27 | 6 | 110 |
| sqn | 7 | 3 | 38 | x6dn | 39 | 5 | 82 |
| sqr6 | 6 | 12 | 50 | xor5 | 5 | 1 | 16 |
| wim | m | 7 | 9 | life_min | 9 | 1 | 84 |

# Appendix D

# Results

## D.1 Results of Three-Level Decomposition Experiments

We first provide a brief overview of the results of our three-level decomposition tool.

- 3level_decomp found 59 benchmarks with XOR logic.
- AOXMIN-MV found 54 benchmarks with XOR logic.
- For 32 benchmarks AOXMIN-MV did not identify XOR logic while 3level_decomp did.
- For 27 benchmarks AOXMIN-MV did identify XOR logic while 3level_decomp did not.
- The total number of benchmarks for which the AOXMIN-MV tool completed without error was 244. Examination of the provided code indicates that a programming error is the cause of the 34 failures.
- The total number of benchmarks for which 3level_decomp completed without error was 278.
- The average time for 3level_decomp to complete was 5.4 seconds while the average time for AOXMIN-MV was 71.1 seconds.
- All benchmarks used had fewer than 10 inputs due to limitations of the proof-

of-concept autocorrelation-based tool.

In the following table, the second and third columns refer to the results of our three-level minimization tool. The column labeled "XOR?" refers to whether or not the algorithm identified the presence of XOR logic in the function; a "no" indicates no appropriate three-level decomposition, while a "yes" indicates that one was found. The column labeled "3level_decomp & ac time" gives the timing in seconds for computation of the autocorrelation coefficients and determining the decomposition. The third and fourth columns give similar results for the comparison tool AOXMIN-MV. The entry "err" means that for some reason the tool in question could not complete processing for that benchmark.

| filename | 3LEVEL | | AOXMIN-MV | | filename | 3LEVEL | | AOXMIN-MV | |
|---|---|---|---|---|---|---|---|---|---|
| | XOR? | time | XOR? | time | | XOR? | time | XOR? | time |
| dc1_out7 | no | 0 | no | 0.0 | adr4_out1 | yes | 0 | no | 0.0 |
| 5xp1_out1 | no | 0 | no | 0.2 | adr4_out2 | yes | 0 | yes | 2.1 |
| 5xp1_out10 | no | 0 | no | 0.0 | adr4_out3 | yes | 0 | yes | 1.7 |
| 5xp1_out2 | no | 0 | no | 0.2 | adr4_out4 | yes | 0 | yes | 1.9 |
| 5xp1_out3 | yes | 0 | yes | 4.4 | adr4_out5 | yes | 0 | no | 0.2 |
| 5xp1_out4 | yes | 0 | yes | 3.6 | alu2_out1 | no | 6 | yes | 11.1 |
| 5xp1_out5 | yes | 0 | yes | 2.3 | alu2_out2 | no | 24 | yes | 1.4 |
| 5xp1_out6 | yes | 0 | yes | 1.8 | alu2_out3 | yes | 0 | no | 0.3 |
| 5xp1_out7 | yes | 0 | yes | 0.1 | alu2_out4 | yes | 0 | no | 0.0 |
| 5xp1_out8 | yes | 0 | no | 0.2 | alu2_out5 | no | 18 | no | 0.2 |
| 5xp1_out9 | yes | 0 | no | 0.0 | alu2_out6 | yes | 0 | no | 0.1 |
| 9symml_out1 | no | 6 | err | err | alu3_out1 | yes | 0 | yes | 0.3 |
| C17_out1 | no | 0 | no | 0.0 | alu3_out2 | no | 0 | yes | 0.2 |
| C17_out2 | no | 0 | err | err | alu3_out3 | no | 6 | yes | 0.3 |
| C7552_out1 | no | 0 | no | 0.0 | alu3_out4 | no | 6 | no | 0.1 |
| C7552_out10 | no | 0 | no | 0.0 | alu3_out5 | no | 6 | no | 0.0 |
| C7552_out11 | no | 0 | no | 0.0 | alu3_out6 | no | 12 | yes | 0.3 |
| C7552_out12 | no | 0 | no | 0.0 | alu3_out7 | no | 0 | no | 0.0 |
| C7552_out13 | no | 0 | no | 0.0 | alu3_out8 | no | 0 | no | 0.0 |
| C7552_out14 | no | 0 | no | 0.0 | apex4_out1 | no | 0 | no | 0.0 |
| C7552_out15 | no | 0 | no | 0.0 | apex4_out2 | no | 0 | no | 0.2 |
| C7552_out16 | no | 0 | no | 0.0 | apex4_out3 | no | 6 | no | 0.5 |
| C7552_out2 | no | 0 | no | 0.0 | apex4_out4 | no | 6 | no | 0.5 |
| C7552_out3 | no | 0 | no | 0.0 | apex4_out5 | no | 6 | no | 0.5 |
| C7552_out4 | no | 0 | no | 0.0 | apex4_out6 | no | 6 | yes | 4:28.6 |
| C7552_out5 | no | 0 | no | 0.0 | apex4_out7 | no | 6 | no | 0.5 |
| C7552_out6 | no | 0 | no | 0.0 | apex4_out8 | no | 6 | yes | 57.1 |
| C7552_out7 | no | 0 | no | 0.0 | apex4_out9 | no | 6 | no | 0.4 |
| C7552_out8 | no | 0 | no | 0.0 | apex4_out10 | no | 6 | no | 0.5 |
| C7552_out9 | no | 0 | no | 0.0 | apex4_out11 | no | 6 | no | 0.4 |

| filename | 3LEVEL | | AOXMIN-MV | | filename | 3LEVEL | | AOXMIN-MV | |
|---|---|---|---|---|---|---|---|---|---|
| | XOR? | time | XOR? | time | | XOR? | time | XOR? | time |
| apex4_out12 | no | 6 | no | 0.5 | bw_out24 | no | 0 | no | 0.1 |
| apex4_out13 | no | 6 | no | 0.4 | bw_out25 | no | 0 | err | err |
| apex4_out14 | no | 6 | no | 0.3 | bw_out26 | no | 0 | no | 0.2 |
| apex4_out15 | no | 6 | no | 0.4 | bw_out27 | no | 0 | no | 0.2 |
| apex4_out16 | no | 0 | no | 0.0 | bw_out28 | no | 0 | no | 0.0 |
| apex4_out17 | no | 0 | no | 0.0 | bw_out3 | no | 0 | no | 0.2 |
| apex4_out18 | no | 0 | no | 0.0 | bw_out4 | no | 0 | yes | 0.6 |
| apex4_out19 | no | 0 | no | 0.0 | bw_out5 | no | 0 | no | 0.1 |
| apla_out1 | no | 6 | no | 0.2 | bw_out6 | no | 0 | err | err |
| apla_out10 | no | 12 | no | 0.3 | bw_out7 | no | 0 | no | 0.1 |
| apla_out11 | no | 0 | no | 0.0 | bw_out8 | no | 0 | no | 0.0 |
| apla_out12 | no | 0 | no | 0.2 | bw_out9 | no | 0 | yes | 0.7 |
| apla_out2 | no | 6 | no | 0.1 | clip_out1 | no | 0 | no | 0.0 |
| apla_out3 | no | 6 | no | 0.0 | clip_out2 | no | 0 | yes | 1.6 |
| apla_out4 | no | 0 | no | 0.0 | clip_out3 | no | 6 | yes | 3.7 |
| apla_out5 | no | 6 | no | 0.1 | clip_out4 | no | 6 | yes | 1.6 |
| apla_out6 | no | 0 | no | 0.0 | clip_out5 | no | 0 | no | 0.0 |
| apla_out7 | no | 0 | no | 0.0 | cm42a_out1 | no | 0 | err | err |
| apla_out8 | no | 0 | no | 0.1 | cm42a_out10 | no | 0 | err | err |
| apla_out9 | no | 6 | no | 0.1 | cm42a_out2 | no | 0 | err | err |
| bw_out1 | no | 0 | no | 0.1 | cm42a_out3 | no | 0 | err | err |
| bw_out10 | no | 0 | no | 0.0 | cm42a_out4 | no | 0 | err | err |
| bw_out11 | no | 0 | no | 0.1 | cm42a_out5 | no | 0 | err | err |
| bw_out12 | no | 0 | no | 0.1 | cm42a_out6 | no | 0 | err | err |
| bw_out13 | no | 0 | no | 0.0 | cm42a_out7 | no | 0 | err | err |
| bw_out14 | no | 0 | no | 0.1 | cm42a_out8 | no | 0 | err | err |
| bw_out15 | no | 0 | no | 0.1 | cm42a_out9 | no | 0 | err | err |
| bw_out16 | no | 0 | no | 0.0 | cm82a_out1 | yes | 0 | yes | 1.6 |
| bw_out17 | no | 0 | no | 0.0 | cm82a_out2 | yes | 0 | yes | 2.7 |
| bw_out18 | no | 0 | no | 0.1 | cm82a_out3 | yes | 0 | no | 0.0 |
| bw_out19 | no | 0 | no | 0.1 | con1_out1 | no | 0 | no | 0.2 |
| bw_out2 | no | 0 | no | 0.0 | con1_out2 | no | 0 | no | 0.4 |
| bw_out20 | no | 0 | yes | 0.7 | dc1_out1 | yes | 0 | no | 0.1 |
| bw_out21 | yes | 0 | yes | 1.8 | dc1_out2 | no | 0 | no | 0.1 |
| bw_out22 | no | 0 | no | 0.0 | dc1_out3 | no | 0 | yes | 0.7 |
| bw_out23 | no | 0 | yes | 1.4 | dc1_out4 | no | 0 | yes | 0.7 |

| filename | 3LEVEL | | AOXMIN-MV | | filename | 3LEVEL | | AOXMIN-MV | |
|---|---|---|---|---|---|---|---|---|---|
| | XOR? | time | XOR? | time | | XOR? | time | XOR? | time |
| dc1_out5 | yes | 0 | no | 0.2 | dk17_out5 | no | 0 | no | 0.1 |
| dc1_out6 | yes | 0 | no | 0.0 | dk17_out6 | no | 0 | no | 0.0 |
| dc2_out1 | no | 0 | no | 0.3 | dk17_out7 | no | 0 | no | 0.0 |
| dc2_out2 | no | 0 | no | 0.1 | dk17_out8 | no | 0 | no | 0.0 |
| dc2_out3 | no | 0 | no | 0.2 | dk17_out9 | no | 6 | no | 0.0 |
| dc2_out4 | no | 0 | no | 0.1 | dk27_out1 | no | 0 | no | 0.1 |
| dc2_out5 | no | 0 | yes | 0.9 | dk27_out2 | no | 0 | no | 0.0 |
| dc2_out6 | no | 0 | no | 0.2 | dk27_out3 | no | 0 | no | 0.1 |
| dc2_out7 | yes | 0 | no | 0.0 | dk27_out4 | yes | 0 | no | 0.0 |
| decod_out1 | no | 0 | no | 0.0 | dk27_out5 | no | 0 | no | 0.0 |
| decod_out10 | no | 0 | no | 0.0 | dk27_out6 | yes | 0 | no | 0.0 |
| decod_out11 | no | 0 | no | 0.0 | dk27_out7 | yes | 0 | no | 0.0 |
| decod_out12 | no | 0 | no | 0.0 | dk27_out8 | no | 0 | no | 0.0 |
| decod_out13 | no | 0 | no | 0.0 | dk27_out9 | yes | 0 | no | 0.0 |
| decod_out2 | no | 0 | no | 0.0 | ex1_out1 | yes | 0 | yes | 39.1 |
| decod_out14 | no | 0 | no | 0.0 | ex2_out1 | no | 0 | no | 0.3 |
| decod_out15 | no | 0 | no | 0.0 | ex3_out1 | no | 0 | no | 0.1 |
| decod_out16 | no | 0 | no | 0.0 | example2_out1 | no | 12 | yes | 10.8 |
| decod_out3 | no | 0 | no | 0.0 | example2_out2 | no | 24 | yes | 1.5 |
| decod_out4 | no | 0 | no | 0.0 | example2_out3 | yes | 0 | no | 0.2 |
| decod_out5 | no | 0 | no | 0.0 | example2_out4 | yes | 0 | no | 0.0 |
| decod_out6 | no | 0 | no | 0.0 | example2_out5 | no | 18 | no | 0.2 |
| decod_out7 | no | 0 | no | 0.0 | example2_out6 | yes | 0 | no | 0.1 |
| decod_out8 | no | 0 | no | 0.0 | f2_out1 | no | 0 | no | 0.0 |
| decod_out9 | no | 0 | no | 0.0 | f2_out2 | no | 0 | no | 0.0 |
| dist_out1 | no | 0 | no | 0.0 | f2_out3 | no | 0 | no | 0.0 |
| dist_out2 | no | 0 | yes | 1.5 | f2_out4 | no | 0 | no | 0.0 |
| dist_out3 | no | 0 | yes | 7.7 | life_min_out1 | no | 6 | yes | 5:16.5 |
| dist_out4 | no | 0 | no | 0.3 | life_out1 | no | 6 | yes | 4:49.1 |
| dist_out5 | no | 0 | no | 0.2 | log8mod_out1 | no | 0 | no | 0.3 |
| dk17_out1 | no | 0 | no | 0.1 | log8mod_out2 | no | 0 | no | 0.2 |
| dk17_out10 | no | 6 | no | 0.2 | log8mod_out3 | no | 0 | yes | 1.1 |
| dk17_out11 | no | 0 | no | 0.1 | log8mod_out4 | no | 0 | no | 0.2 |
| dk17_out2 | no | 0 | no | 0.1 | log8mod_out5 | no | 0 | yes | 2.3 |
| dk17_out3 | no | 0 | no | 0.0 | majority_out1 | no | 0 | no | 0.0 |
| dk17_out4 | no | 6 | no | 0.1 | max46_out1 | no | 6 | no | 0.3 |

| filename | 3LEVEL | | AOXMIN-MV | | filename | 3LEVEL | | AOXMIN-MV | |
|---|---|---|---|---|---|---|---|---|---|
| | XOR? | time | XOR? | time | | XOR? | time | XOR? | time |
| misex1_out1 | no | 0 | no | 0.1 | sqn_out3 | no | 0 | yes | 1.2 |
| misex1_out2 | no | 0 | no | 0.1 | sqr6_out1 | no | 0 | no | 0.0 |
| misex1_out3 | no | 0 | no | 0.1 | sqr6_out10 | yes | 0 | no | 0.0 |
| misex1_out4 | no | 0 | no | 0.1 | sqr6_out11 | no | 0 | no | 0.0 |
| misex1_out5 | yes | 0 | no | 0.2 | sqr6_out12 | yes | 0 | no | 0.0 |
| misex1_out6 | no | 0 | no | 0.2 | sqr6_out2 | no | 0 | no | 0.1 |
| misex1_out7 | no | 0 | no | 0.2 | sqr6_out4 | no | 0 | yes | 1.0 |
| mlp4_out1 | no | 0 | no | 0.0 | sqr6_out3 | no | 0 | no | 0.2 |
| mlp4_out2 | no | 0 | no | 0.2 | sqr6_out5 | no | 0 | no | 0.1 |
| mlp4_out3 | no | 0 | no | 0.3 | sqr6_out6 | yes | 0 | no | 0.1 |
| mlp4_out4 | no | 0 | no | 0.3 | sqr6_out7 | yes | 0 | yes | 1.9 |
| mlp4_out5 | yes | 0 | yes | 18.1 | sqr6_out8 | yes | 0 | no | 0.2 |
| mlp4_out6 | yes | 0 | yes | 4.6 | sym10_out1 | no | 24 | err | err |
| mlp4_out7 | yes | 0 | yes | 0.2 | sqr6_out9 | yes | 0 | no | 0.1 |
| mlp4_out8 | yes | 0 | no | 0.0 | wim_out1 | no | 0 | err | err |
| pm1_out4 | no | 0 | err | err | wim_out2 | no | 0 | no | 0.0 |
| pm1_out1 | no | 0 | err | err | wim_out3 | yes | 0 | no | 0.1 |
| pm1_out10 | no | 0 | err | err | wim_out4 | no | 0 | no | 0.0 |
| pm1_out2 | no | 0 | err | err | wim_out5 | no | 0 | err | err |
| pm1_out3 | no | 0 | err | err | wim_out6 | no | 0 | err | err |
| pm1_out5 | no | 0 | err | err | wim_out7 | no | 0 | err | err |
| pm1_out6 | no | 0 | err | err | x2_out1 | no | 0 | err | err |
| pm1_out7 | no | 0 | err | err | x2_out2 | yes | 0 | no | 0.1 |
| pm1_out8 | no | 0 | err | err | x2_out3 | no | 0 | no | 0.0 |
| pm1_out9 | no | 0 | err | err | x2_out4 | no | 0 | err | err |
| radd_out1 | yes | 0 | no | 0.2 | x2_out5 | no | 0 | no | 0.0 |
| radd_out2 | yes | 0 | yes | 2.1 | x2_out6 | no | 12 | no | 0.2 |
| radd_out3 | yes | 0 | yes | 2.1 | x2_out7 | no | 12 | no | 0.1 |
| radd_out4 | yes | 0 | yes | 1.8 | xor5_out1 | yes | 0 | yes | 39.2 |
| radd_out5 | yes | 0 | no | 0.0 | z4_out1 | yes | 0 | no | 0.0 |
| ryy6_out1 | no | 2640 | err | err | z4_out2 | yes | 0 | yes | 2.0 |
| sao2_out1 | no | 6 | no | 0.1 | z4_out3 | yes | 0 | yes | 2.7 |
| sao2_out2 | no | 6 | no | 0.2 | z4_out4 | yes | 0 | yes | 1.8 |
| sao2_out3 | no | 12 | err | err | z4ml_out1 | yes | 0 | no | 0.0 |
| sao2_out4 | no | 6 | err | err | z4ml_out2 | yes | 0 | yes | 1.8 |
| sqn_out1 | no | 0 | no | 0.1 | z4ml_out3 | yes | 0 | yes | 2.7 |
| sqn_out2 | no | 0 | no | 0.2 | z4ml_out4 | yes | 0 | yes | 1.7 |

## D.2 Results of KDD Ordering and Decomposition

The following is a brief overview of some of the results of comparing our KDD dtl and ordering tool with the DTL_SIFT algorithm implemented in the PUMA package. Our method is referred to as AC.

- The AC method resulted in a smaller KDD than did the DLT_SIFT algorithm for 4 benchmarks.
- The Sift method (DTL_SIFT algorithm) resulted in a smaller KDD for 131 benchmarks.
- Both methods resulted in KDDs of the same size for 141 benchmarks.
- If a one node size difference is permitted then AC and Sift result in "same-sized" KDDs for 173 benchmarks.
- The average nodes for the AC method is 16.5, while the average nodes for the Sift method is 14.0.
- Both methods completed successfully for 276 benchmarks.
- The average time for the Sift method was 0.09 seconds while the average time for the AC method was 0.43 seconds.

The complete results are listed in the following tables. In these tables the time in seconds required for computing the autcorrelation coefficients, determining the dtl and variable ordering, and building the kdd is listed in the second column labeled "AC method/time". The resulting number of nodes for the KDD is then listed in the third column labeled "AC method/nodes". The corresponding measures for the DTL_SIFT algorithm are listed in columns 4 and 5.

| filename | AC method | | DTL_SIFT | |
|---|---|---|---|---|
| | time | nodes | time | nodes |
| 5xp1_out1 | 0 | 11 | 0.1 | 10 |
| 5xp1_out10 | 0 | 7 | 0.1 | 7 |
| 5xp1_out2 | 0.1 | 19 | 0.1 | 13 |
| 5xp1_out3 | 0 | 16 | 0.1 | 12 |
| 5xp1_out4 | 0.1 | 12 | 0.1 | 9 |
| 5xp1_out5 | 0 | 7 | 0.1 | 7 |
| 5xp1_out6 | 0 | 5 | 0 | 4 |
| 5xp1_out7 | 0 | 3 | 0 | 3 |
| 5xp1_out8 | 0.1 | 2 | 0.1 | 2 |
| 5xp1_out9 | 0 | 1 | 0.1 | 1 |
| 9symml_out1 | 0.2 | 24 | 0.1 | 24 |
| C17_out1 | 0 | 6 | 0.1 | 5 |
| C17_out2 | 0 | 6 | 0.1 | 4 |
| C7552_out1 | 0 | 5 | 0.1 | 5 |
| C7552_out10 | 0 | 5 | 0.1 | 5 |
| C7552_out11 | 0 | 5 | 0.1 | 5 |
| C7552_out12 | 0 | 5 | 0.1 | 5 |
| C7552_out13 | 0 | 5 | 0 | 5 |
| C7552_out14 | 0.1 | 5 | 0.1 | 5 |
| C7552_out2 | 0.1 | 5 | 0.1 | 5 |
| C7552_out15 | 0 | 5 | 0.1 | 5 |
| C7552_out16 | 0 | 5 | 0.1 | 5 |
| C7552_out3 | 0 | 5 | 0 | 5 |
| C7552_out4 | 0.1 | 5 | 0 | 5 |
| C7552_out5 | 0 | 5 | 0.1 | 5 |
| C7552_out6 | 0 | 5 | 0 | 5 |
| C7552_out7 | 0 | 5 | 0.1 | 5 |
| C7552_out8 | 0 | 5 | 0.1 | 5 |
| C7552_out9 | 0.1 | 5 | 0.1 | 5 |
| adr4_out1 | 0 | 11 | 0.1 | 11 |
| adr4_out2 | 0.1 | 10 | 0.1 | 11 |
| adr4_out3 | 0 | 7 | 0.1 | 8 |
| adr4_out4 | 0 | 4 | 0.1 | 4 |
| adr4_out5 | 0 | 2 | 0.1 | 2 |
| alu2_out1 | 0.3 | 46 | 0.1 | 33 |

| filename | AC method | | DTL_SIFT | |
|---|---|---|---|---|
| | time | nodes | time | nodes |
| alu2_out2 | 1 | 86 | 0.1 | 47 |
| alu2_out3 | 0 | 2 | 0.1 | 2 |
| alu2_out4 | 0.1 | 2 | 0.1 | 2 |
| alu2_out5 | 0.6 | 81 | 0.1 | 64 |
| alu2_out6 | 0.1 | 6 | 0 | 4 |
| alu3_out1 | 0.2 | 4 | 0 | 4 |
| alu3_out2 | 0.1 | 10 | 0.1 | 7 |
| alu3_out3 | 0.2 | 15 | 0.1 | 10 |
| alu3_out4 | 0.4 | 55 | 0.1 | 24 |
| alu3_out5 | 0.2 | 9 | 0.1 | 9 |
| alu3_out6 | 0.5 | 20 | 0.1 | 14 |
| alu3_out7 | 0.1 | 7 | 0.1 | 7 |
| alu3_out8 | 0.1 | 4 | 0.1 | 4 |
| apex4_out1 | err | err | err | err |
| apex4_out10 | 0.2 | 103 | 0.1 | 98 |
| apex4_out11 | 0.4 | 100 | 0.1 | 97 |
| apex4_out12 | 0.2 | 90 | 0.1 | 81 |
| apex4_out13 | 0.2 | 80 | 0.1 | 73 |
| apex4_out14 | 0.3 | 80 | 0.1 | 73 |
| apex4_out15 | 0.2 | 80 | 0.1 | 82 |
| apex4_out16 | 0.1 | 30 | 0.1 | 25 |
| apex4_out17 | 0.2 | 26 | 0.1 | 23 |
| apex4_out18 | 0.1 | 28 | 0.1 | 25 |
| apex4_out19 | 0.1 | 33 | 0.1 | 25 |
| apex4_out2 | 0.2 | 70 | 0.1 | 61 |
| apex4_out3 | 0.4 | 105 | 0.1 | 94 |
| apex4_out4 | 0.2 | 105 | 0.1 | 92 |
| apex4_out5 | 0.4 | 106 | 0.1 | 95 |
| apex4_out6 | 0.2 | 105 | 0.1 | 93 |
| apex4_out7 | 0.4 | 98 | 0.2 | 97 |
| apex4_out8 | 0.3 | 102 | 0.2 | 95 |
| apex4_out9 | 0.3 | 106 | 0.1 | 99 |
| apla_out1 | 0.2 | 20 | 0.1 | 14 |
| apla_out10 | 0.4 | 34 | 0 | 23 |
| apla_out11 | 0.2 | 10 | 0.1 | 9 |
| apla_out12 | 0.2 | 17 | 0.1 | 10 |

| filename | AC method | | DTL_SIFT | |
|---|---|---|---|---|
| | time | nodes | time | nodes |
| apla_out2 | 0.3 | 20 | 0.1 | 14 |
| apla_out3 | 0.3 | 20 | 0.1 | 16 |
| apla_out4 | 0.1 | 8 | 0.1 | 8 |
| apla_out5 | 0.4 | 24 | 0 | 18 |
| apla_out6 | 0.1 | 8 | 0.1 | 8 |
| apla_out7 | 0.2 | 4 | 0.1 | 4 |
| apla_out8 | 0.1 | 23 | 0.1 | 15 |
| apla_out9 | 0.2 | 14 | 0.1 | 10 |
| bw_out1 | 0 | 12 | 0.1 | 9 |
| bw_out10 | 0 | 5 | 0.1 | 5 |
| bw_out11 | 0 | 7 | 0.1 | 5 |
| bw_out12 | 0 | 8 | 0.1 | 6 |
| bw_out13 | 0 | 5 | 0.1 | 5 |
| bw_out14 | 0 | 10 | 0.1 | 8 |
| bw_out15 | 0 | 9 | 0 | 7 |
| bw_out16 | 0 | 7 | 0.1 | 6 |
| bw_out17 | 0 | 5 | 0 | 5 |
| bw_out18 | 0.1 | 7 | 0.1 | 7 |
| bw_out19 | 0.1 | 7 | 0.1 | 6 |
| bw_out2 | 0 | 5 | 0.1 | 5 |
| bw_out20 | 0 | 9 | 0.1 | 7 |
| bw_out21 | 0.1 | 6 | 0.1 | 5 |
| bw_out22 | 0 | 5 | 0.1 | 5 |
| bw_out23 | 0.1 | 9 | 0.1 | 7 |
| bw_out24 | 0.1 | 10 | 0.1 | 9 |
| bw_out25 | 0 | 8 | 0.1 | 7 |
| bw_out26 | 0 | 9 | 0.1 | 8 |
| bw_out27 | 0 | 8 | 0.1 | 6 |
| bw_out28 | 0 | 5 | 0.1 | 5 |
| bw_out3 | 0.1 | 8 | 0 | 6 |
| bw_out4 | 0.1 | 6 | 0.1 | 6 |
| bw_out5 | 0 | 10 | 0.1 | 8 |
| bw_out6 | 0 | 8 | 0.1 | 7 |
| bw_out7 | 0 | 9 | 0.1 | 9 |
| bw_out8 | 0.1 | 7 | 0 | 5 |
| bw_out9 | 0.1 | 10 | 0.1 | 7 |

| filename | AC method | | DTL_SIFT | |
|---|---|---|---|---|
| | time | nodes | time | nodes |
| clip_out1 | 0.2 | 32 | 0.1 | 26 |
| clip_out2 | 0.2 | 32 | 0.1 | 27 |
| clip_out3 | 0.2 | 39 | 0.1 | 28 |
| clip_out4 | 0.2 | 27 | 0.1 | 21 |
| clip_out5 | 0.1 | 20 | 0.1 | 20 |
| cm42a_out1 | 0 | 4 | 0.1 | 4 |
| cm42a_out10 | 0 | 4 | 0.1 | 4 |
| cm42a_out2 | 0 | 4 | 0.1 | 4 |
| cm42a_out3 | 0.1 | 4 | 0.1 | 4 |
| cm42a_out4 | 0 | 4 | 0.1 | 4 |
| cm42a_out5 | 0 | 4 | 0.1 | 4 |
| cm42a_out6 | 0.1 | 4 | 0.1 | 4 |
| cm42a_out7 | 0.1 | 4 | 0.1 | 4 |
| cm42a_out8 | 0 | 4 | 0.1 | 4 |
| cm42a_out9 | 0.1 | 4 | 0.1 | 4 |
| cm82a_out1 | 0.1 | 3 | 0.1 | 3 |
| cm82a_out2 | 0 | 6 | 0.1 | 6 |
| cm82a_out3 | 0 | 7 | 0 | 7 |
| con1_out1 | 0 | 10 | 0 | 9 |
| con1_out2 | 0 | 8 | 0.1 | 6 |
| dc1_out1 | 0 | 6 | 0.1 | 5 |
| dc1_out2 | 0 | 4 | 0.1 | 4 |
| dc1_out3 | 0 | 6 | 0.1 | 4 |
| dc1_out4 | 0 | 6 | 0.1 | 5 |
| dc1_out5 | 0.1 | 7 | 0.1 | 6 |
| dc1_out6 | 0 | 6 | 0.1 | 5 |
| dc1_out7 | 0 | 6 | 0 | 4 |
| dc2_out1 | 0 | 10 | 0 | 8 |
| dc2_out2 | 0 | 17 | 0.1 | 15 |
| dc2_out3 | 0.1 | 20 | 0.1 | 16 |
| dc2_out4 | 0 | 26 | 0.1 | 19 |
| dc2_out5 | 0 | 20 | 0.1 | 10 |
| dc2_out6 | 0 | 11 | 0.1 | 8 |
| dc2_out7 | 0.1 | 1 | 0 | 1 |
| decod_out1 | 0 | 5 | 0 | 5 |
| decod_out10 | 0 | 5 | 0.1 | 5 |

| filename | AC method | | DTL_SIFT | | filename | AC method | | DTL_SIFT | |
|---|---|---|---|---|---|---|---|---|---|
| | time | nodes | time | nodes | | time | nodes | time | nodes |
| decod_out11 | 0.1 | 5 | 0.1 | 5 | dk27_out7 | 0.1 | 2 | 0.1 | 2 |
| decod_out12 | 0 | 5 | 0.1 | 5 | dk27_out8 | 0 | 6 | 0.1 | 6 |
| decod_out13 | 0.1 | 5 | 0.1 | 5 | dk27_out9 | 0 | 1 | 0.1 | 1 |
| decod_out14 | 0 | 5 | 0.1 | 5 | ex1_out1 | 0 | 5 | 0 | 5 |
| decod_out2 | 0 | 5 | 0.1 | 5 | ex2_out1 | 0 | 10 | 0.1 | 8 |
| decod_out15 | 0.1 | 5 | 0.1 | 5 | ex3_out1 | 0 | 9 | 0 | 8 |
| decod_out16 | 0 | 5 | 0.1 | 5 | example2_out1 | 0.4 | 46 | 0.1 | 33 |
| decod_out3 | 0 | 5 | 0.1 | 5 | example2_out2 | 1 | 86 | 0.1 | 47 |
| decod_out4 | 0 | 5 | 0.1 | 5 | example2_out3 | 0.1 | 2 | 0.1 | 2 |
| decod_out5 | 0 | 5 | 0.1 | 5 | example2_out4 | 0.1 | 2 | 0.1 | 2 |
| decod_out6 | 0 | 5 | 0.1 | 5 | example2_out5 | 0.7 | 81 | 0.1 | 64 |
| decod_out7 | 0.1 | 5 | 0.1 | 5 | example2_out6 | 0.2 | 6 | 0.1 | 4 |
| decod_out8 | 0.1 | 5 | 0.1 | 5 | f2_out1 | 0.1 | 5 | 0.1 | 5 |
| decod_out9 | 0 | 5 | 0.1 | 5 | f2_out2 | 0.1 | 5 | 0.1 | 5 |
| dist_out1 | 0 | 23 | 0.1 | 20 | f2_out3 | 0 | 5 | 0 | 5 |
| dist_out2 | 0 | 28 | 0.1 | 23 | f2_out4 | 0 | 5 | 0.1 | 5 |
| dist_out3 | 0.1 | 37 | 0.1 | 32 | life_out1 | 0.2 | 26 | 0.1 | 25 |
| dist_out4 | 0.1 | 55 | 0.1 | 36 | life_min_out1 | 0.3 | 26 | 0.1 | 25 |
| dist_out5 | 0.1 | 58 | 0.1 | 46 | log8mod_out1 | 0 | 9 | 0.1 | 6 |
| dk17_out1 | 0.1 | 9 | 0.1 | 9 | log8mod_out2 | 0 | 13 | 0.1 | 13 |
| dk17_out10 | 0.2 | 17 | 0.1 | 12 | log8mod_out3 | 0.1 | 15 | 0.1 | 13 |
| dk17_out11 | 0.2 | 9 | 0.1 | 8 | log8mod_out4 | 0 | 22 | 0.1 | 17 |
| dk17_out2 | 0.2 | 9 | 0.1 | 9 | log8mod_out5 | 0.1 | 25 | 0.1 | 19 |
| dk17_out3 | 0.1 | 10 | 0.1 | 9 | majority_out1 | 0 | 7 | 0 | 7 |
| dk17_out4 | 0.3 | 16 | 0.1 | 11 | max46_out1 | 0.2 | 80 | 0.1 | 74 |
| dk17_out5 | 0.2 | 15 | 0.1 | 10 | misex1_out1 | 0 | 6 | 0.1 | 5 |
| dk17_out6 | 0.1 | 3 | 0.1 | 3 | misex1_out2 | 0.1 | 11 | 0.1 | 8 |
| dk17_out7 | 0.1 | 3 | 0 | 3 | misex1_out3 | 0 | 12 | 0.1 | 10 |
| dk17_out8 | 0.1 | 3 | 0.1 | 3 | misex1_out4 | 0.1 | 12 | 0.1 | 9 |
| dk17_out9 | 0.2 | 18 | 0.1 | 13 | misex1_out5 | 0 | 6 | 0.1 | 5 |
| dk27_out1 | 0 | 3 | 0.1 | 3 | misex1_out6 | 0 | 12 | 0.1 | 9 |
| dk27_out2 | 0 | 3 | 0.1 | 3 | misex1_out7 | 0.1 | 12 | 0.1 | 9 |
| dk27_out3 | 0 | 7 | 0.1 | 7 | mlp4_out1 | 0 | 14 | 0.1 | 14 |
| dk27_out4 | 0 | 2 | 0 | 2 | mlp4_out2 | 0.1 | 27 | 0.1 | 25 |
| dk27_out5 | 0.1 | 5 | 0.1 | 5 | mlp4_out3 | 0 | 37 | 0.1 | 36 |
| dk27_out6 | 0.1 | 2 | 0 | 2 | mlp4_out4 | 0.1 | 49 | 0.1 | 39 |

| filename | AC method | | DTL_SIFT | |
|---|---|---|---|---|
| | time | nodes | time | nodes |
| mlp4_out5 | 0.1 | 21 | 0.1 | 15 |
| mlp4_out6 | 0 | 10 | 0.1 | 7 |
| mlp4_out7 | 0 | 4 | 0 | 4 |
| mlp4_out8 | 0 | 2 | 0.1 | 2 |
| pm1_out1 | 0 | 4 | 0.1 | 4 |
| pm1_out10 | 0.1 | 4 | 0.1 | 4 |
| pm1_out2 | 0 | 4 | 0.1 | 4 |
| pm1_out3 | 0.1 | 4 | 0.1 | 4 |
| pm1_out4 | 0.1 | 4 | 0.1 | 4 |
| pm1_out5 | 0.1 | 4 | 0.1 | 4 |
| pm1_out6 | 0 | 4 | 0.1 | 4 |
| pm1_out7 | 0 | 4 | 0.1 | 4 |
| pm1_out8 | 0 | 4 | 0.1 | 4 |
| pm1_out9 | 0.1 | 4 | 0.1 | 4 |
| radd_out1 | 0 | 2 | 0.1 | 2 |
| radd_out2 | 0.1 | 4 | 0.1 | 4 |
| radd_out3 | 0 | 7 | 0.1 | 7 |
| radd_out4 | 0.1 | 10 | 0.1 | 10 |
| radd_out5 | 0 | 11 | 0.1 | 11 |
| ryy6_out1 | 92.2 | 31 | 0.1 | 21 |
| sao2_out1 | 0.2 | 31 | 0.1 | 29 |
| sao2_out2 | 0.3 | 49 | 0.1 | 33 |
| sao2_out3 | 0.5 | 36 | 0.1 | 24 |
| sao2_out4 | 0.3 | 35 | 0.1 | 25 |
| sqn_out1 | 0 | 25 | 0.1 | 13 |
| sqn_out2 | 0.1 | 31 | 0.1 | 20 |
| sqn_out3 | 0 | 26 | 0.1 | 25 |
| sqr6_out1 | 0 | 5 | 0 | 5 |
| sqr6_out10 | 0 | 2 | 0.1 | 2 |
| sqr6_out11 | err | err | err | err |
| sqr6_out12 | 0 | 1 | 0.1 | 1 |
| sqr6_out2 | 0 | 6 | 0.1 | 5 |

| filename | AC method | | DTL_SIFT | |
|---|---|---|---|---|
| | time | nodes | time | nodes |
| sqr6_out3 | 0 | 13 | 0.1 | 10 |
| sqr6_out4 | 0.1 | 16 | 0.1 | 12 |
| sqr6_out5 | 0.1 | 15 | 0.1 | 13 |
| sqr6_out6 | 0 | 12 | 0.1 | 12 |
| sqr6_out7 | 0 | 8 | 0 | 6 |
| sqr6_out8 | 0 | 7 | 0 | 4 |
| sqr6_out9 | 0 | 4 | 0 | 3 |
| sym10_out1 | 0.9 | 30 | 0.1 | 30 |
| wim_out1 | 0.1 | 4 | 0.1 | 4 |
| wim_out2 | 0 | 5 | 0.1 | 5 |
| wim_out3 | 0.1 | 3 | 0 | 3 |
| wim_out4 | 0.1 | 3 | 0.1 | 3 |
| wim_out5 | 0 | 3 | 0.1 | 3 |
| wim_out6 | 0 | 3 | 0.1 | 3 |
| wim_out7 | 0 | 6 | 0.1 | 5 |
| x2_out1 | 0 | 3 | 0.1 | 3 |
| x2_out2 | 0.1 | 3 | 0.1 | 3 |
| x2_out3 | 0 | 3 | 0.1 | 3 |
| x2_out4 | 0.1 | 6 | 0.1 | 6 |
| x2_out5 | 0.1 | 4 | 0.1 | 4 |
| x2_out6 | 0.5 | 13 | 0.1 | 14 |
| x2_out7 | 0.5 | 14 | 0.1 | 11 |
| xor5_out1 | 0 | 5 | 0.1 | 5 |
| z4_out1 | 0 | 10 | 0.1 | 10 |
| z4_out2 | 0.1 | 9 | 0.1 | 9 |
| z4_out3 | 0 | 6 | 0.1 | 6 |
| z4_out4 | 0 | 3 | 0.1 | 3 |
| z4ml_out1 | 0 | 10 | 0.1 | 10 |
| z4ml_out2 | 0 | 9 | 0.1 | 9 |
| z4ml_out3 | 0 | 6 | 0.1 | 6 |
| z4ml_out4 | 0.1 | 3 | 0.1 | 3 |

# VITA

| | | | |
|---|---|---|---|
| ***Surname:*** | Rice | ***Given Names:*** | Jacqueline Elsie |
| ***Place of Birth:*** | Canada | ***Date of Birth:*** | Jan. 27, 1972 |

### *Educational Institutions Attended*

| | |
|---|---|
| University of Victoria | 1989 to 1993 |
| University of Victoria | 1993 to 1995 |

### *Degrees Awarded*

| | | |
|---|---|---|
| B.Sc. | University of Victoria | 1993 |
| M.Sc. | University of Victoria | 1995 |

### *Honors and Awards*

| | |
|---|---|
| UVic Graduate Research and Teaching Fellowship | 1993-1995, 1998-2002 |
| Eileen Ford Wood and Alexander James Wood Scholarship | 1999 |
| Canada Scholarship, University of Victoria | 1989-1990 |
| University of Victoria Entrance Scholarship | 1989 |
| International Union of Operating Engineers Closed Scholarship | 1989 |

### *Conference Publications*

**1.** Rice, J., "Autocorrelation Coefficients in the Representation and Classification of Switching Functions", *to be presented at* EDAA Ph.D. Forum at the Design, Automation and Test in Europe Conference, Munich, Germany,. Mar. 3-7 2003.

**2.** Rice, J. and Muzio, J. C., "Use of the Autocorrelation Function in the Classification of Switching Functions", Proceedings of the Euromicro Symposium on Digital System Design: Architectures, Methods and Tools (DSD) 2002 pp. 244-251.

**3.** Rice, J. and Muzio, J. C., "Antisymmetries in the Realization of Boolean Functions", Proceedings of the International Symposium on Circuits and Systems (ISCAS) 2002, paper number 2666.

**4.** Rice, J., Serra, M. and Muzio, J. C., "Methods for Calculating Autocorrelation coefficients", Proceedings of the 4th Internation Workshop on Boolean Problems

(IWSBP) 2000, pp. 69-76.

5. Rice, J., Serra, M. and Muzio, J. C., "The Use of Autocorrelation Coefficients for Variable Ordering for ROBDDs", in the Proceedings of the 4th International Workshop on Applications of the Reed-Muller Expansion in Circuit Design (RM) 1999, pp. 185-196.

### *Work Submitted for Publication*

1. Rice, J., "On the Use of Autocorrelation Coefficients in the Identification of Three-Level Decompositions", *submitted to* the International Workshop on Logic Synthesis, 2003.
2. Rice, J., Kent, K. and Evans, P., "A Reconfigurable Solution for Common Approximate Substring Matching", *submitted to* the 6th International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS) 2003.
3. Rice, J. and Kent, K., "A Hardware Implementation for the Computation of the Autocorrelation Function", *submitted to* the 14th IEEE International Workshop on Rapid System Prototyping (RSP) 2003.
4. Rice, J. and Kent, K., "Using Instance Specific Circuits to Compute Autocorrelation Coefficients", *submitted to* the First Northeast Workshop on Circuits and Systems (NEWCAS) 2003.

Title of Dissertation:

AUTOCORRELATION COEFFICIENTS IN THE REPRESENTATION AND CLASSIFICATION OF SWITCHING FUNCTIONS

Author: ________________________________

JACQUELINE ELSIE RICE
27 Feb. 2003

# DISSERTATION WITHHOLDING FORM

At our request, the commencement of the period for which the partial licence shall operate shall be delayed from 27 Feb. 2003 for a period of at least six months.

________________________________________

(Supervisor)

________________________________________

(Department Chairman)

________________________________________

(Dean of Graduate Studies)

________________________________________

(Signature of Author)

________________________________________

(Date)

Date Submitted to the Dean's Office: ________________________