ESOP CIRCUIT MINIMIZATION BASED ON THE FUNCTION ON-SET

By

Likai Chai

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Electrical Engineering
in the Department of Electrical and Computer Engineering

Mississippi State, Mississippi

August 2000

ESOP CIRCUIT MINIMIZATION BASED ON

THE FUNCTION ON-SET

By

Likai Chai

Approved:

Mitchell A. Thornton Associate Professor of Electrical and Computer Engineering (Director of Thesis)	Robert B. Reese Associate Professor of Electrical and Computer Engineering (Committee Member)
James C. Harden Professor of Electrical and Computer Engineering (Committee Member)	Nicolas H. Younan Graduate Coordinator of the Department Electrical and Computer Engineering
A. Wayne Bennett Dean of the College of Engineering	

Name: Likai Chai

Date of Degree: August 5, 2000

Institution: Mississippi State University

Major Field: Electrical Engineering

Major Professor: Dr. Mitchell A. Thornton

Title of Study: ESOP CIRCUIT MINIMIZATION BASED ON THE FUNCTION ON-SET

Pages in Study: 53

Candidate for Degree of Master of Science

A method for minimizing Exclusive-OR Sum of Product (ESOP) forms of Boolean functions based on function on-set information is described. ESOP forms have been proven to require fewer products than the more common Sum of Products (SOP) forms based on the inclusive-OR operator for many classes of functions. Since finding the optimal ESOP form requires the solution of the well-known set-covering problem, heuristic methods are used. This technique is implemented and the results are given and analyzed.

DEDICATION

This thesis is dedicated to my wonderful family for their unconditional love and support. I wish to thank my parents, Xueyi Chai and Xiaolan Zhang, who always encourage me to be my best and always expect my best effort. My father inspired me to become an electrical engineer, and my mother prayed for me through all my academic years. I also wish to give special thanks to my wonderful wife, Yan Luo, who has labored for me and loved me throughout my time in this graduate program. I cannot adequately express the depth of my gratitude for the encouragement and support of my family.

ACKNOWLEDGMENTS

Sincere appreciation is extended to Dr. Mitchell A. Thornton for serving as major professor for my graduate program and for offering assistance, encouragement, and direction in this thesis project.

Appreciation is expressed to Dr. James C. Harden and Dr. Robert B. Reese for serving on my graduate committee.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

A method for minimizing Exclusive-OR Sum of Product (ESOP) forms of Boolean functions based on function on-set information is described. ESOP forms have been proven to require fewer products than the more common Sum of product (SOP) forms based on the inclusive-OR operator for many classes of functions. Since finding the optimal ESOP form requires the solution of the well-known set-covering problem, heuristic methods are used. This technique is implemented and the results are given and analyzed.

Introduction

Since the emergence of VLSI, logic design using computer-based tools has become indispensable. VLSI designs contain too many transistors for manual implementation. Thus, the importance of logic synthesis has increased greatly [1]. In logic synthesis, minimization methods that were developed many years earlier have recently become practical.

The minimization of Sum of Product (SOP) forms always plays an important role in logic synthesis. This problem is well understood and there are many good minimization algorithms [2,3,4,5] in existence. It is well known that minimizing

Exclusive-OR Sum of Product (ESOP) forms is a more difficult problem than minimizing SOP forms. However, ESOP forms are of interest because they can often represent functions with fewer product terms than the corresponding SOP form [6,7]. Also, many multilevel circuits based on EXOR elements are more advantageous in terms of area, speed and testability. Error detection circuitry is a typical example where EXOR's are heavily used [8].

Two-level logic network minimization techniques are used to find a representation for a Boolean function that is optimum according to a given cost function. Typical cost functions are the number of product terms in a two-level realization, the total number of literals, or, a combination of both.

With any of these cost functions, the problem of two-level Boolean minimization contains the sub-problem of finding the solution of the minimum set covering problem, which has been shown to be nondeterministic polynomial-time *NP*-complete [9]. Nevertheless, sophisticated exact minimizers have been developed whose average-case behavior for most commonly encountered functions is acceptable [10,11]. Furthermore, heuristic minimization methods exist which have been shown to produce results that are close to the minimum within reasonable amounts of CPU time, even for large Boolean functions [12,13]. The existence of these techniques has enabled the development and widespread use of logic synthesis tools that are prevalent in a modern design environment [1,14,15].

The minimization of a SOP form is equivalent to the problem of covering all the 1-cells in the Karnaugh map at least once by using a minimum number of loops. On the other hand, the minimization of an ESOP is equivalent to the problem of covering all the 1-cells an odd number of times and all the 0-cells either in an even number of times or not at all, by using a minimum number of loops. For example, consider the function $f(w,x,y,z) = \sum (5,7,13,14)$ shown in Fig. 1.1. The map on the left contains circles that result in the minimized SOP form, $f = x\bar{y}z + \bar{w}xz + w\bar{x}yz$. The map on the right contains circles that result in the minimized ESOP form, $f = xz \oplus wyz$.

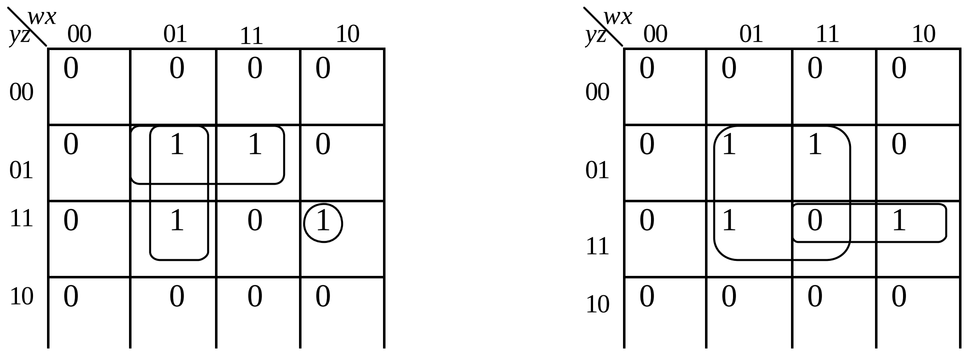


Fig. 1.1. Representation using SOP form versus ESOP form.

SOP versus ESOP

The two-level SOP form corresponds to two level logic networks using only AND and OR operations with the presence of complemented literals assumed. ESOP forms correspond to two-level logic networks where an EXOR is used instead of an OR. Logic networks are often designed by using programmable logic arrays (PLAs) that have an AND-OR structure, as shown in Fig. 1.2a. Because this regular structure of the PLAs can be easily designed, modified and laid out, it is used extensively in modern VLSI design. By replacing the OR gate with the EXOR gate in the PLA, we have an AND-EXOR PLA as shown in Fig. 1.2b. AND-EXOR PLA's have several advantages over AND-OR PLA's [16]. Firstly, AND-EXOR PLA's often require fewer products than AND-OR PLA's. Table 1 compares the number of products of various classes of functions [16] of *n* variables. Secondly, AND-EXOR PLA's are usually easier to test than AND-OR PLA's. Similar to AND-OR PLA's, AND-EXOR PLA's can be made to have universal test sets [17], and since AND-EXOR PLAs generally require a smaller amount of hardware, they can also have a shorter test sequence [18].

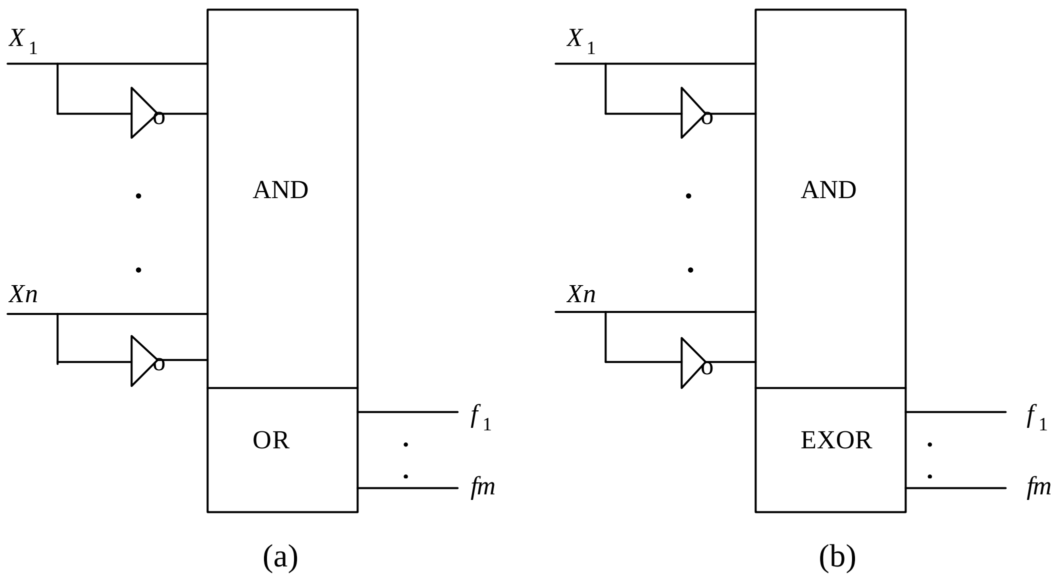


Fig. 1.2. AND-OR PLA versus AND-EXOR PLA.

Although AND-EXOR PLA's have such merits, several problems must be solved before they can be used in practical designs. The first problem is that EXOR's are generally more expensive and slower than OR's. The second problem is that the minimization of AND-EXOR PLA's is more difficult than that of AND-OR PLA's. This work focuses on the latter problem.

TABLE 1 [16]

Number of Products to Realize Various Functions

	AND-OR PLA (SOP)		AND-EXOR PLA (ESOP)	
	1-bit decoders	2-bit decoders	1-bit decoders	2-bit decoders
Arbitrary functions	2^{n-1}	$\frac{1}{2} . 2^{n-1}$	$\frac{1}{2} . 2^{n-1}$	$\frac{1}{2} . 2^{n-1}$
Symmetric functions	2^{n-1}	$\frac{1}{3} . 3^{n/2}$	$\frac{2}{3} . 3^{n/2}$	$\frac{1}{3} . 3^{n/2}$
Parity functions	2^{n-1}	$\frac{1}{2} . 2^{n/2}$	n	$\frac{n}{2}$
n-bit adders	$6 . 2^{n} - 4n-5$	$n^{2} + 1$	$2^{n+1} - 1$	$\frac{1}{2} . (n^{2} + 3n - 2)$

Methods of Minimization

The minimization of SOP’s has been studied for more than 30 years. Many well known methods have been developed. However, the minimization of ESOP’s has not been solved to the same degree.

SOP Minimization Methods

The minimization of a two level logic network is one of the key technologies in logic synthesis. Many well known methods have been developed such as MINI, and ESPRESSO [12,13]. The Karnaugh map method is an *n*-dimensional cube written in a plane and is convenient for the representation of logic functions and can be used to minimize functions with 3-5 variables. The tabulation method (Quine-McCluskey method) [3] can be extended for multi-valued input two-valued output functions. In practical applications, the problems are multi-output in most cases, thus the number of prime implicants is very large. Therefore, the tabulation method can require an excessive amount of memory. Even for cases where all the prime implicants can be generated, if the number of input variables is large, then the number of minterms is also very large causing the cover table to require too much memory for storage. For these reasons, the tabulation method can be unsuitable for large problems.

Several heuristic methods have been developed for minimizing SOPs with many inputs [12,19]. These heuristic methods do not necessarily produce optimal solutions, but they can achieve fairly good solutions in a short time. In practical applications, it is not necessary to obtain an exact minimum solution since a near minimum solution may be sufficient. Thus, these heuristic methods are extensively used for SOPs in modern technologies. Among these heuristic methods, MINI developed at IBM, and ESPRESSO developed at IBM and UC Berkeley are well known.

However, if the number of input variables is small, the tabulation method is faster than other heuristic methods such as MINI and ESPRESSO. Also, the tabulation method always obtains the exact minimum solutions. Therefore, tabulation method is useful for theoretical research. Recently, the tabulation method was implemented utilizing the Binary Decision Diagram (BDD) [20] data structure and can obtain exact minimum solutions with many inputs [10].

ESOP Minimization Methods

Several heuristic methods have been developed such as EXORCISM, MINT and EXMIN2 [16,18,21]. These methods use simplification rules and there continues to be much room for improvement. Before using the simplification rules, the initial expression must be transformed into an ESOP expression. Commonly, a disjoint sum-of-products expression (DSOP) is used which can become very large. A DSOP is a SOP where no product has a common minterm.

Helliwell and Perkowski developed EXORCISM at Portland State University. In EXORCISM, two operations are used to link the cubes, the primary xlink and the secondary xlink [18], that generalizes the known operations of merger, exclusion and other logic operations. Both operations can be applied under certain conditions. If two cubes are of the same dimension, then the primary xlink operation can be applied. If the distance of two cubes is 1, the secondary xlink can be applied.

EXMIN2 and MINT are rule based ESOP minimizers. A large set of rules was proposed by Sasao in EXMIN2, and extended by Kozlowski, Dagless and Saul in MINT. In EXMIN2 and MINT, a set of rules are used to link two cubes. Each rule can be applied under certain conditions. For instance, the rule RESHAPE can be applied on two terms $X^A Y^B$ and $X^C Y^D$ if $A \cap C =$ Æ and $B \supset D$ as $X^A Y^{(B \cap \overline{D})}$ and $X^{(A \cup C)} Y^D$ [16]. The number of operations in EXMIN2 and MINT is larger than that in EXORCISM-this is one of the reasons why EXMIN2 and MINT generate better results than EXORCISM. EXMIN2 sometimes produces good solutions although it can be very time-consuming

[16]. Furthermore, exact ESOP minimization can only be accomplished for functions of up to 6 variables [16].

Most ESOP minimizers [16,18,21] use rules that take two or more product terms and transform them to other terms. Minimimizers based on this concept can be slow since applying a single transformation rule to all pairs in a list of product terms is quadratic in the number of product terms.

An alternative heuristic ESOP minimization technique, TABESOP is described in this thesis. The method is derived by modifying the tabulation technique that is used for SOP minimization. There are two advantages of this technique. First, the initial expression does not have to be DSOP. Second, the minimization algorithm is not rule based, so it forms a good basis for experimentation with exact and heuristic methods for solving the ESOP cover problem.

Fig. 1.3 shows the relations among ESOP, TABESOP and DSOP. Functions in SOP form only include minterms from the on-set. Functions in ESOP form also include minterms from the off-set since even numbers of these minterms can be cancelled out. If the initial function is in SOP form, it must be transformed into ESOP form before input to the ESOP minimizer. Usually, the disjoint sharp operation is used to transform the SOP into the DSOP forms. Functions in DSOP form only include minterms from the on-set. TABESOP can transform SOP forms into ESOP forms that only include minterms from the on-set. This ESOP form is generally smaller than a DSOP since the technique is based on the tabulation method.

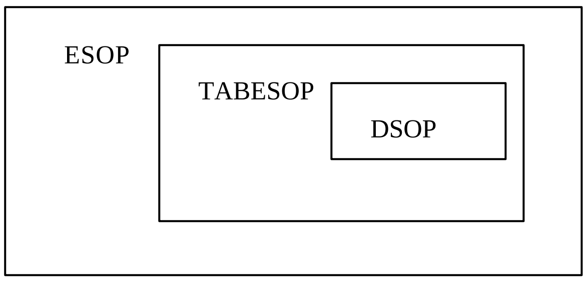


Fig. 1.3.   Relations among ESOP, TABESOP and DSOP.

Motivation and Background

EXMIN2 and MINT are currently the most powerful ESOP minimization tools. However, the current state of the art is still immature compared with techniques for SOP minimization. Many researchers are currently investigating techniques for minimizing ESOPs since the discovery of such a method would have important consequences with regard to the practical problems of logic synthesis, logic verification and error detection.

The motivation of this research is not only to implement a good minimization algorithm for ESOPs, but also to evaluate some theoretical considerations since no exact minimization algorithm for ESOP exists.

Contribution of This Work

In this work, an experimental computer program that operates over a structure similar to the cover table used in the tabular approach for SOP minimization was developed. The technique is implemented using two approaches to solve the set-covering problem. In the first approach, the product term that covers the minimum minterms is chosen first. In the second approach, the product term that covers the maximum minterms is chosen first.

Also, the cover table storage requirements are reduced by using the BDD structure for representation.

Outline

In chapter 2, the heuristic ESOP minimization technique is described. First, the basic concept of this technique is presented. Second, a quick review of the SOP tabulation method is provided. Next, the ESOP-based tabulation method is described in full detail.

In chapter 3, two approaches to solve the cover table are provided. The first approach used is based on minimum minterm counts. The second approach used is based on maximum minterm counts. Examples are given to illustrate these approaches. Also, extensions of the minimization method for handling multi-output functions is described.

In chapter 4, the implementation of this technique is introduced and experimental results are presented that compare this approach with two well-known popular approaches for minimizing ESOP forms.

Chapter 5 presents conclusions and discusses possible further extensions to this approach.

CHAPTER II

EXTENSIONS TO TABULATION METHOD

In this chapter, the heuristic ESOP minimization technique is described. First, the basic concept of this technique is presented. Second, a brief review of the tabulation method is provided. Finally, the actual technique is introduced in full detail.

Basic Concept

An ESOP minimization technique based on odd/even cube covering is described here. This technique is based on two fundamental relations as given in Equations 1 and 2.

$$\bigoplus_{i=1}^{2m} c = 0 \qquad (1)$$

$$\bigoplus_{i=1}^{2m+1} c = c \qquad (2)$$

Where c represents an instance of a specific product term (cube or logic AND) of a set of literals, $\oplus$ represents the exclusive-OR operation and m is some non-negative integer. Equation 1 essentially states that including a cube in a cover set an even number of times is equivalent to not including it at all. Likewise, including a cube in a cover set an odd number of times is equivalent to including it with an instance of one. This gives a degree of freedom in finding a minimal ESOP cover set that does not necessarily mean it is in the overall on-set of the function being minimized.

As an example, consider the function, $f(w,x,y,z) = \sum(0,1,4,7,10,11,13,14)$. Fig. 2.1 shows two Karnaugh maps. Fig. 2.2 shows two corresponding networks that realize the function. The map on the left contains circles that result in the minimized SOP form, $f = \bar{w}\,\bar{x}\,\bar{y} + \bar{w}\,\bar{y}\,\bar{z} + \bar{w}xyz + wx\bar{y}z + wy\bar{z} + w\bar{x}y$. The realization of SOP form requires 6 AND gates and the overall expression is comprised of 20 literals. The Karnaugh map on the right contains circles that obey Equations 1 and 2 since all 0 values are circled an even number of times and all 1 values are circled an odd number of times. Thus, the 0 values cancel out due to the XOR relation in Equation 1 and the 1 values remain. This results in the expression, $f = \bar{w}\,\bar{y} \oplus xz \oplus wy$ which only requires 3 AND gates and is comprised of 6 literals.

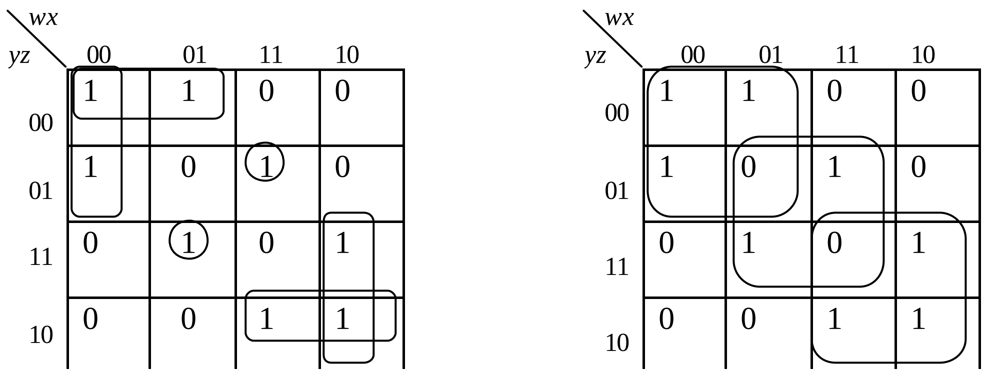


Fig. 2.1. Karnaugh map illustrating SOP form versus ESOP form.

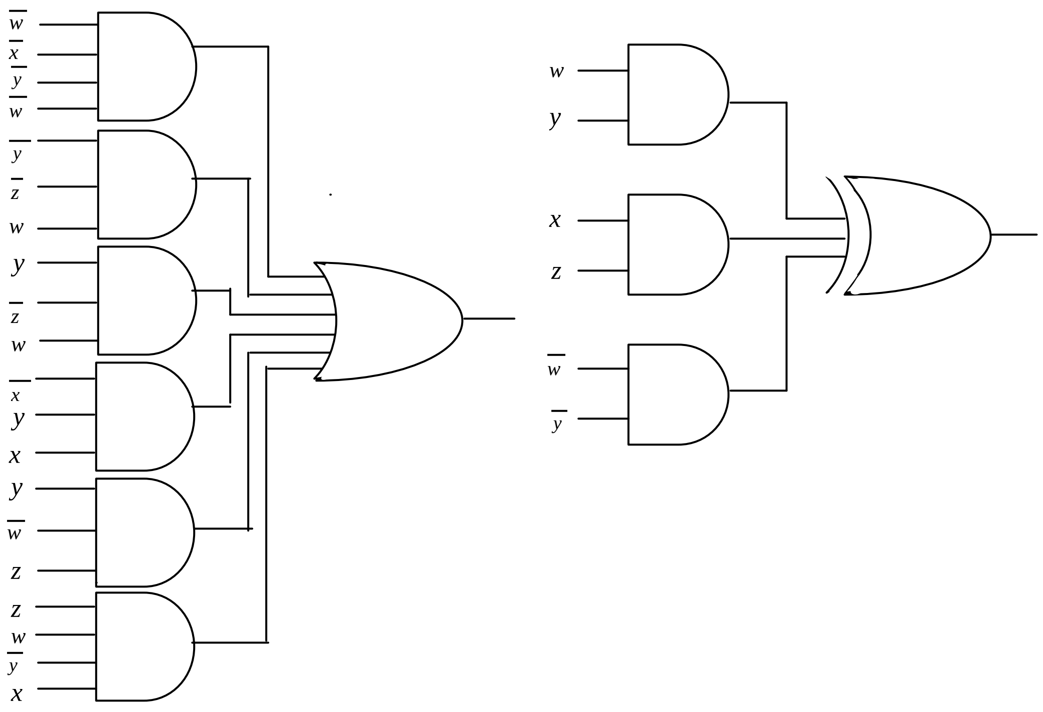


Fig. 2.2.   AND-OR realization versus AND-EXOR realization.

Tabulation Method

The tabulation method is the first algorithmic method proposed for two-level minimization that includes prime implicant generation, implicant table generation and a cover table solving technique. It can be summarized as the following two steps:

1. Generation of the set of prime implicants for a given function.
2. Selection of a minimum set of prime implicants to implement the function.

Prime Implicants

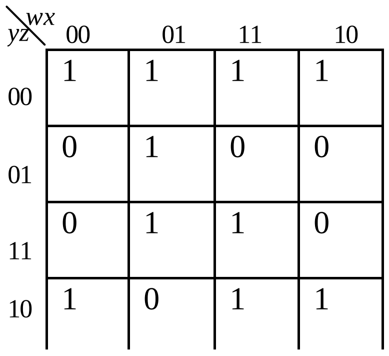
The tabulation method is best explained here with an example. A Boolean function is represented as a minterm list and a Karnaugh map is shown in Fig. 2.3a. Each pair of minterms is compared to see if they differ by only one literal and can be merged into a single cube. The cubes generated in this example by pairing common minterms are shown in Fig. 2.3b. These cubes are called *1*-cubes because they contain exactly one "don't care". A minterm is thus a *0*-cube. In the next step, the *1*-cubes are checked to see if they can be merged into *2*-cubes. The *2*-cubes generated from *1*-cubes are shown in Fig. 2.3c. If a *k*-cube is generated by pairing two *(k-1)*-cubes, then the two *(k-1)*-cubes are not prime implicants and can be discarded. The process is continued until no pairing is possible in the final cube list.

Prime Implicant Table

The prime implicants of the function in Fig. 2.3c have been marked as *A, B, C,* D, *E, F* and *G* . We now build a prime implicant table as shown in Fig. 2.4. The rows of the prime implicant table are the minterms, and the columns are the prime implicants. An "*X*" in the prime implicant table in row *i* and column *j* means that the minterm corresponding to row *i* is contained by the prime implicant corresponding to column *j*. For example, the minterm 0001 is contained by prime implicant *A*, 0-01.

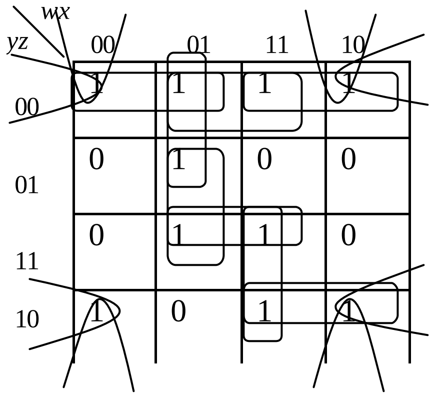
In the prime implicant table, we have to choose a minimum set of prime implicants (columns) such that there is at least one *X* in every row. This is an instance of the set-

0000	1011
0001	1101
0010	1111
1000	
0011	
0101	
1010	



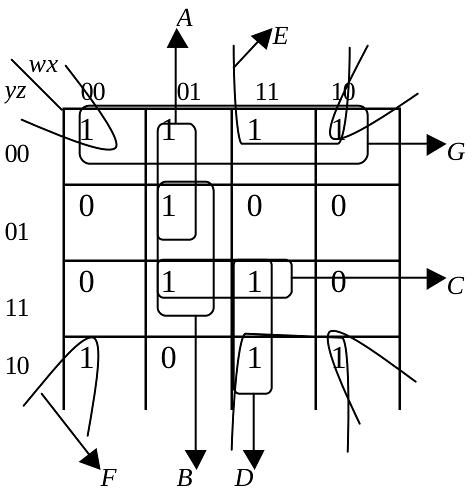
(a)

000-	1-11
00-1	101-
001-	-000
00-0	10-0
0-01	-010
-101	-011
11-1	



(b)

0-01	*A*
-101	*B*
11-1	*C*
1-11	*D*
-01-	*E*
-0-0	*F*
00--	*G*



(c)

Fig. 2.3. Prime implicant generation.

covering problem. This problem has been shown to be *NP*-complete [9].

	A	*B*	*C*	*D*	*G*	*F*	*E*
0000					x	x	
0001	x				x		
0010					x	x	x
0011					x		x
0101	x	x					
1000						x	
1010						x	x
1011				x			x
1101		x	x				
1111			x	x			

Fig. 2.4. Prime implicant table.

Essential Prime Implicants

A row with a single *X* means that the corresponding prime implicant is an essential prime implicant. Any prime cover for the function will have to contain the prime implicant that contains the minterm corresponding to this row, because this minterm is not contained by any other prime implicant (column). In the prime implicant table of Fig. 2.4, *F* is the essential prime implicant.

Dominated Columns

We obtain the reduced table of Fig. 2.5a after deleting column *F* and four rows contained by *F*. In the table of Fig. 2.5a, there is no row with a single *X* and no column covers more minterms than another. It is necessary to use some type of rule for choosing the next prime implicant. Assume that prime implicant A is selected. We obtain the reduced table of Fig. 2.5b after deleting column *A* and two rows contained by *A*. A row *i* of a prime implicant table is said to dominate another row *j* if *I* has an *X* in every column in which *j* has an *X*. In the reduced table of Fig. 2.5b column *B* is dominated by column *C*, and column *G* is dominated by column *E*. We can delete the dominated columns, since selecting the dominating column will result in covering more uncontained minterms than dominated column. Reducing the table of Fig. 2.5b results in the table of Fig. 2.5c. In this table *C* and *E* are relatively essential prime implicants. Choosing *C* and *E* results in *f = {A, C, E, F}*. The final result compares with the initial minterm list as shown in Fig. 2.6.

	A	*B*	*C*	*D*	*G*	*E*
0001	x				x	
0011					x	x
0101	x	x				
1011				x		x
1101		x	x			
1111			x	x		

(a)

	B	*C*	*D*	*G*	*E*
0011				x	x
1011			x		x
1101	x	x			
1111		x	x		

(b)

	C	*D*	*E*
0011			x
1011		x	x
1101	x		
1111	x	x	

(c)

Fig. 2.5. Cyclic prime implicant table.

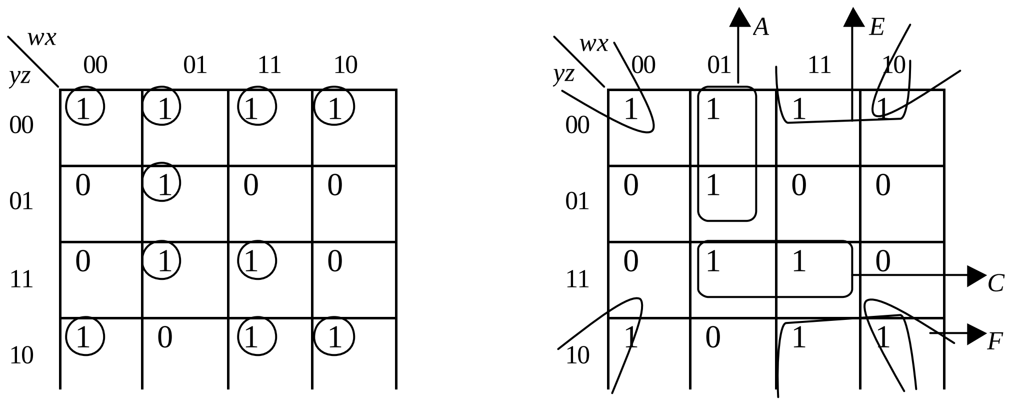


Fig. 2.6. Initial minterm list versus final result.

ESOP Minimization Based on Tabulation Method

The ESOP minimization technique is very similar to the tabulation method where a set of minterms describing the function to be minimized is used as the initial input. It is summarized in the following three steps:

1. Pairs of *(k-1)*-cubes are examined to determine if they differ by only a single literal. If they do, these two *(k-1)*-cubes can be merged into one *k*-cube and new *k*-cube is added to the list. This process is then applied to the new set of cubes and is continued until no new product terms may be added.
2. The list of cubes resulting from step1 is reduced in size by eliminating redundant product terms. This is the step where the cover problem is solved through the use of heuristics.

3. The reduced cube list is further optimized by searching for pairs that satisfy particular algebraic properties. When such a pair is found, they may be replaced by another pair containing more “don’t care” values.

With the exception of step 3, the method outlined above could be interpreted as a high-level description of the tabulation method for SOP circuits. However, due to the different properties of the inclusive-OR versus the exclusive-OR operation, each step must implemented in a slightly different manner.

In step 1, after new cubes are added to the list, the pair of minterms that produced each new cube cannot automatically be deleted as is the case with the SOP tabulation method. Rather, a count of the total number of cubes in the new list must be made that cover the original set. Only if the count is an odd number, {1,3,5,…}, may the original cube be deleted. This is due to the relationships in Equations 1 and 2. Furthermore, when a new cube is created, it is added to the list only if it does not exist there already. This is important since the occurrence of two identical cubes would cause a cancellation since $x \oplus x = 0$. For the SOP formulation, this is not a problem since $x + x = x$.

The second phase of the process involves the creation of a cover table followed by solving the cover problem. This is also done in a similar manner to the SOP tabulation method where “essential products” are chosen for the final list. However, as was the case in step 1, care must be taken to ensure that all minterms are covered an odd number of times, thus the odd/even counting process is also integrated into this procedure.

In the event that the essential cubes do not cover all minterms, a situation occurs that is referred to as a “cyclic cover”. This is a case where there is no clear choice in which product term to add to the final list because none of the candidates are essential and it is an instance of the cover problem. In this work, two ways are used to solve the cover table. The first heuristic method employed is to choose the product term that covers the fewest minterms. The second heuristic method employed is to choose the product term that covers the most minterms. The minimum minterm count method results in potentially adding fewer minterms to cover table during odd/even counting. The maximum minterm count method can cause more minterms to be added to cover table, but allows more rows to be deleted.

After step 2, an ESOP is represented by the resulting cube list that is guaranteed to cover each minterm of f an odd number of times and each minterm of $\overline{f}$ exactly zero times. However, as was shown in the Karnaugh map example, it is permissible to cover minterms of $\overline{f}$ an even number of times, since these function zeros will cancel due to XOR operation. Furthermore, it is often desirable to include these zeros in the resultant ESOP since it can help to result in a smaller list of larger cubes.

Step 3 makes use of three algebraic properties, $1 \oplus x = \overline{x}$, $1 \oplus \overline{x} = x$, and $x \oplus \overline{x} = 1$. Each pair of cubes in the list produced in step 2 is checked to determine if they satisfy the algebraic properties. The first two properties cause three cubes to be reduced into two cubes. The last property allows the two cubes to be merged into a single cube.

Example of Generation of the Cover Table

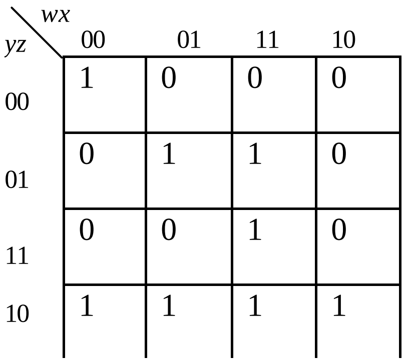
This technique is illustrated with an example. A single-output Boolean function represented as a cube list and Karnaugh map is shown in Fig. 2.7a. The first step of the example is illustrated in the following section. And the second step of the example is illustrated in chapter 3.

Pairing Generation and Odd/Even Cover Checking

The first step of the technique includes two parts. The first part involves pairing generation. And the second part involves odd/even cover checking. The first part of step 1 is the same as the tabulation method, cube merging. Each pair of *(k-1)*-cubes is checked to see if they differ by only one literal. If they do, these two *(k-1)*-cubes can be merged into one *k*-cube and new *k*-cube is added to the list. This process is then applied to the new set of cubes and is continued until no new cubes may be added. In this example, the *1*-cubes generated from the *0*-cubes and the *2*-cubes generated from the *1*-cubes are shown in Fig. 2.7 b and 2.7c, respectively.

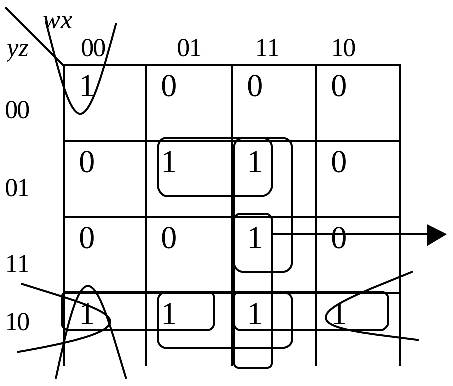
The second part of step 1 involves odd/even cover checking, which differs from the SOP tabulation method. Each minterm is checked to see whether or not they are covered an odd number of times by the product terms in the list. In this example, minterms “0111”, “1000”, “1011” and “1111” are covered twice. Thus, these minterms can not automatically be deleted as would be the case with the SOP tabulation method. We have to add them into the new list to ensure that all minterms are covered an odd number of times as shown in Fig. 2.8.

0000
0101
0111
1000
1001
1010
1011
1111



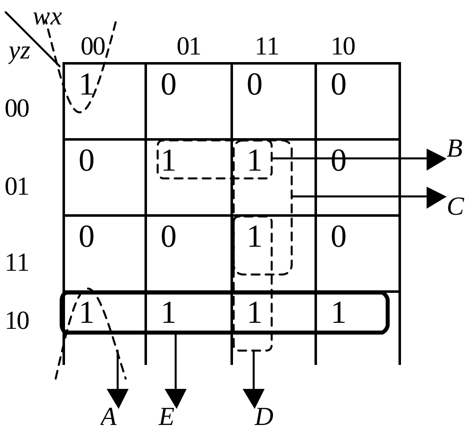
(a)

-000
01-1
-111
1-11
100-
10-0
10-1
101-



(b)

-000	*A*
01-1	*B*
-111	*C*
1-11	*D*
10--	*E*



(c)

Fig. 2.7. Step1- pairing generation for single-output function.

-000	*A*	**1011**	***I***
01-1	*B*		
-111	*C*		
1-11	*D*		
10--	*E*		
0111	***F***		
1000	***G***		
1111	***H***		

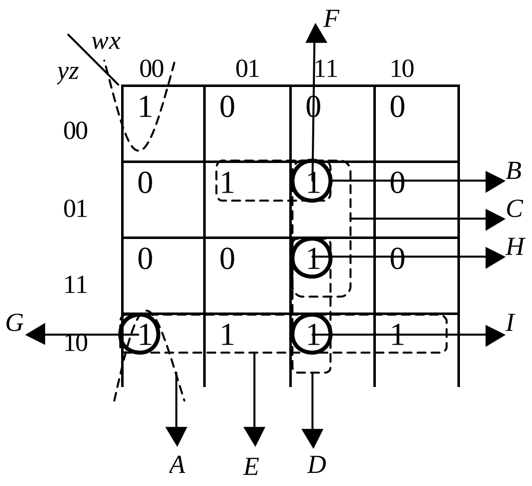


Fig. 2.8. Step1- odd/even cover checking for single-output function.

CHAPTER III

FINDING THE MINIMUM ESOP COVER

In this chapter, Two approaches to solve the cover table are provided. The first approach used is based on minimum minterm counts. The second approach used is based on maximum minterm counts. Examples are given to illustrate these approaches. Also, this chapter presents a minimization method for multi-output functions.

Single-output Functions

A single-output Boolean function represented as a cube list and Karnaugh map is shown in Fig. 2.7a. The new cube list generated after first step is shown in Fig. 2.8.

Cover Table Solving – Cube List Approach

The second step of the technique also includes two parts. The first part involves solving the cover table problem. And the second part is odd/even cover checking. The cube list of the function in Fig. 2.8 is marked as *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H* and *I*. Next, a cover table is built as shown in Fig. 3.1. The rows of the cover table correspond to the minterms of the original cube list, and the columns are the cubes generated from step 1. The number of "*X*"s in each row must be an odd number since each minterm must be covered odd number of times after first step.

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*
0000	x								
0101		x							
0111		x	x			x			
1000	x				x		x		
1001					x				
1010					x				
1011				x	x				x
1111			x	x				x	

Fig. 3.1. Step 2 - cover table for single-output function.

Solving the cover table is also done in a similar manner to the SOP tabulation method where "essential products" are chosen for the final list. A row with a single *X* means that this cube is an essential cube. In the cover table of Fig. 3.3, *A, B* and *E* are the essential cubes. The reduced cover table shown in Fig. 3.2 is obtained after deleting columns *A* , *B*, *E* and the seven rows covered by *A*, *B* and *E*.

	C	*D*	*F*	*G*	*H*	*I*
1111	x	x			x	

Fig. 3.2. Reduced cover table for single-output function.

In the cover table shown in Fig. 3.2, there is no row with a single *X*, and the essential cubes do not cover all minterms. In this technique, Two approaches are used to solve the cover table problem. The first method is to choose the product term that covers the fewest minterms. In this example *H* covers the fewest minterms. Choosing *H* results in *f* = *{A, B, E, H}*. And the second method is to choose the product term that covers the maximum minterms. In this example *C* covers the maximum minterms. Choosing *C* results in *f* = *{A, B, E, C}*. The final results by using minimum minterm counts and maximum minterm counts methods are shown in Fig. 3.3.

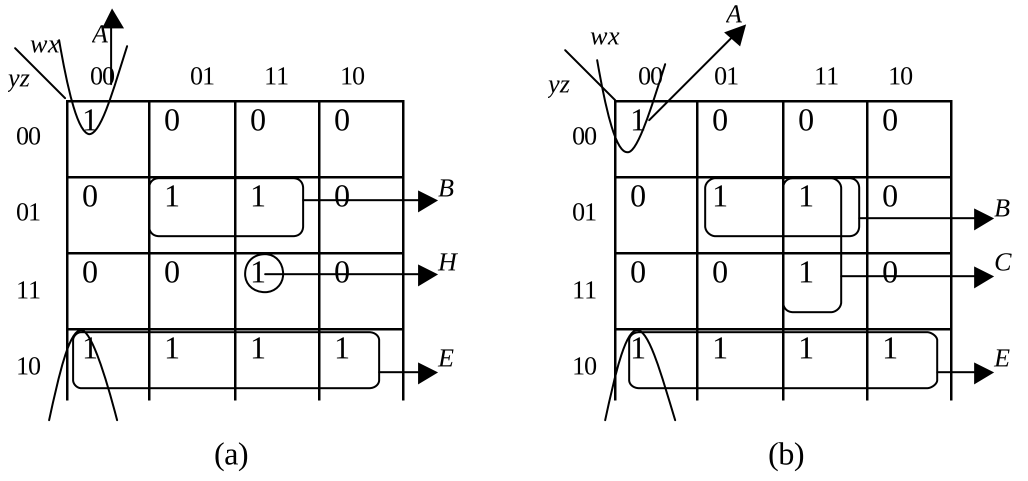


Fig. 3.3. Minimum minterm counts versus maximum minterm counts.

The two heuristic methods to solve the cover table problem are based on the fact that each time an implicant is added to the final list, the odd/even counting process must be invoked. The minimum minterm count method results in potentially adding fewer minterms to cover table during odd/even counting. The maximum minterm count method can cause more minterms to be added to cover table, but allows more rows to be deleted.

As was the case in step 1, the second part of step 2 must be taken to ensure that all minterms are covered an odd number of times. Thus, the odd/even counting process is also integrated into this procedure. In Fig. 3.3a, minterm “1000” is covered twice. So, minterm “1000” cannot be deleted, and is added into the new list. The final result by using minimum minterm count method is shown in Fig. 3.4. In Fig. 3.3b, minterm “0111” and “1000” are covered twice. Thus, they cannot be deleted and are added into the new list. The final result by using maximum minterm count method is shown in Fig. 3.5.

-000	*A*
01-1	*B*
10--	*E*
1111	*H*
1000	

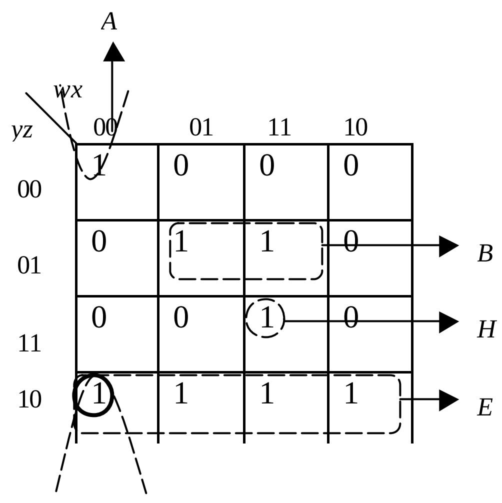


Fig. 3.4. Final result – minimum minterm counts for single-output function.

From this example we can see that five product terms result from using the minimum minterm count method and six product terms by using the maximum minterm count method. The minimum minterm count method produces a better result in this example. However, the maximum minterm count method sometimes produces better results for other circuits. This case is shown in Fig. 3.6. In this example, the minimum minterm count method produces four product terms, and the maximum minterm count method produces three product terms.

-000	*A*
01-1	*B*
-111	*C*
10--	*E*
1000	
0111	

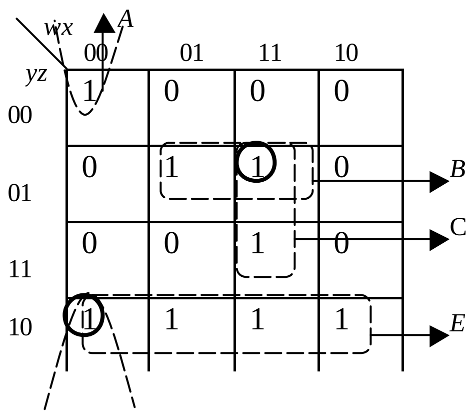


Fig. 3.5. Final result – maximum minterm counts for single-output function.

<u>Cover Table Solving – BDD Approach</u>

The cover table can be solved by using Petrick's method [22]. In the cover table of Fig. 3.1, minterm "0000" is covered when A = 1. Similarly, for "0101", …, "1111", we have the following condition:

0101 : $B = 1$,

0111 : $B + C + F = 1$,

1000 : $A + E + G = 1$,

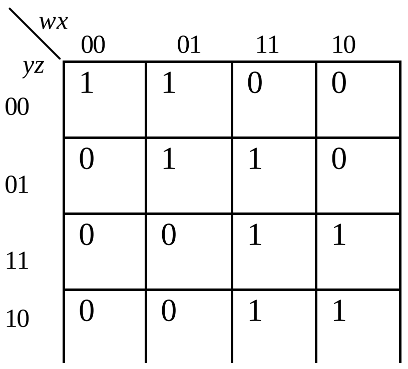
1001 : $E = 1$,

1010 : $E = 1$,

1011 : $D + E + I = 1$,

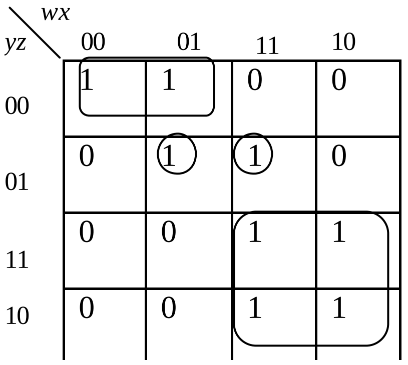
1111 : $C + D + H = 1$.

0000 1111
0001
0101
0111
1010
1011
1110



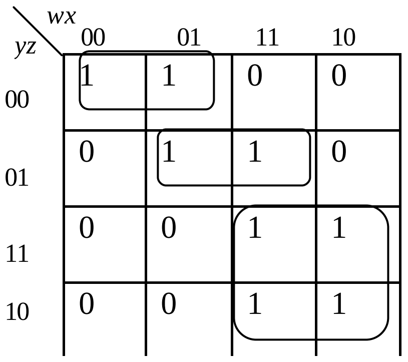
(a) Initial cube list.

000-
0101
0111
1-1-



(b) Final result by using minimum minterm count method.

000-
01-1
1-1-



(c) Final result by using maximum minterm count method.

Fig. 3.6. Maximum minterm count method produces better result.

These conditions are satisfied simultaneously if and only if:

$$AB(B+C+F)(A+E+G)E(D+E+I)(C+D+H) = 1.$$

Due to each minterm needing to be covered an odd number of times, the inclusive-OR is replaced with the exclusive-OR operator:

$$AB(B\oplus C\oplus F)(A\oplus E\oplus G)E(D\oplus E\oplus I)(C\oplus D\oplus H) = 1$$

Unfortunately the solution to this equation is very difficult since it is binate whereas for the SOP cover, it is unate. This is an area for further research.

A binary decision diagram (BDD) is a graphical representation of a logic function, and often has a more compact representation than other methods. Thus, we can utilize BDD data structure to represent the cover table.

For example, consider the logic function $f = A+BC$. Let $A = 1$, then we have $f = 1$. Let $A = 0$, if $B = 0$, then $f = 0$. Let $A = 0$ and $B = 1$. In this case, if $C = 1$ then $f = 1$, and if $C = 0$, then $f = 0$. Fig. 3.7 shows this relation as a decision graph, where $\textcircled{A}$, $\textcircled{B}$, and $\textcircled{C}$ denote non-terminal nodes, and correspond to the input variables. Also, $\boxed{0}$ and $\boxed{1}$

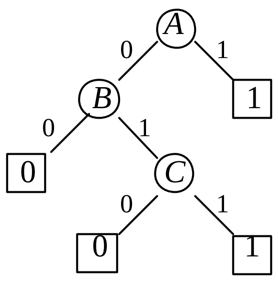


Fig. 3.7. BDD for $f = A + BC$.

are terminal nodes, and correspond to the values of the logic function. Such a decision diagram is a BDD.

To transform the logic expression into BDD, we apply the following Shannon expansion repeatedly:

$$f(A,B,C,\ldots) = \overline{A} f(0, B, C, \ldots) + A f(1, B, C, \ldots).$$

Fig. 3.8 shows the BDD of the example function

$$f = AB(B+C+F)(A+E+G)E(D+E+I)(C+D+H)$$

by applying the Shanon expansion. First, in f, set $A = 0$ to obtain f_0. And write f_1 as the right child of the node A, and obtain Fig. 3.8a. Do the similar operations to variable B, to obtain the two sub-functions f_{10}, f_{11} in Fig. 3.8b. Next, similar operations are performed for the rest of the variables. In this case, if the same sub-functions appear, then the nodes are shared. Fig. 3.8c shows the complete BDD.

Multi-output functions

For multiple-output problems that are too large or complicated to be solved by using Karnaugh map, it is necessary to turn to a tabulation method. The most obvious way to extend the single-output technique would be to form each of the product functions - f_1 , f_2 , …- and then to determine the prime implicants for all the original functions and all the product functions. This is accomplished by forming a single cube for each common cube that appears in any of the output functions. This single cube is made up of two parts: the input portion, which is the same as the cube in the single-output functions,

$$f = AB(B+C+F)(A+E+G)E(D+E+I)(C+D+H)$$

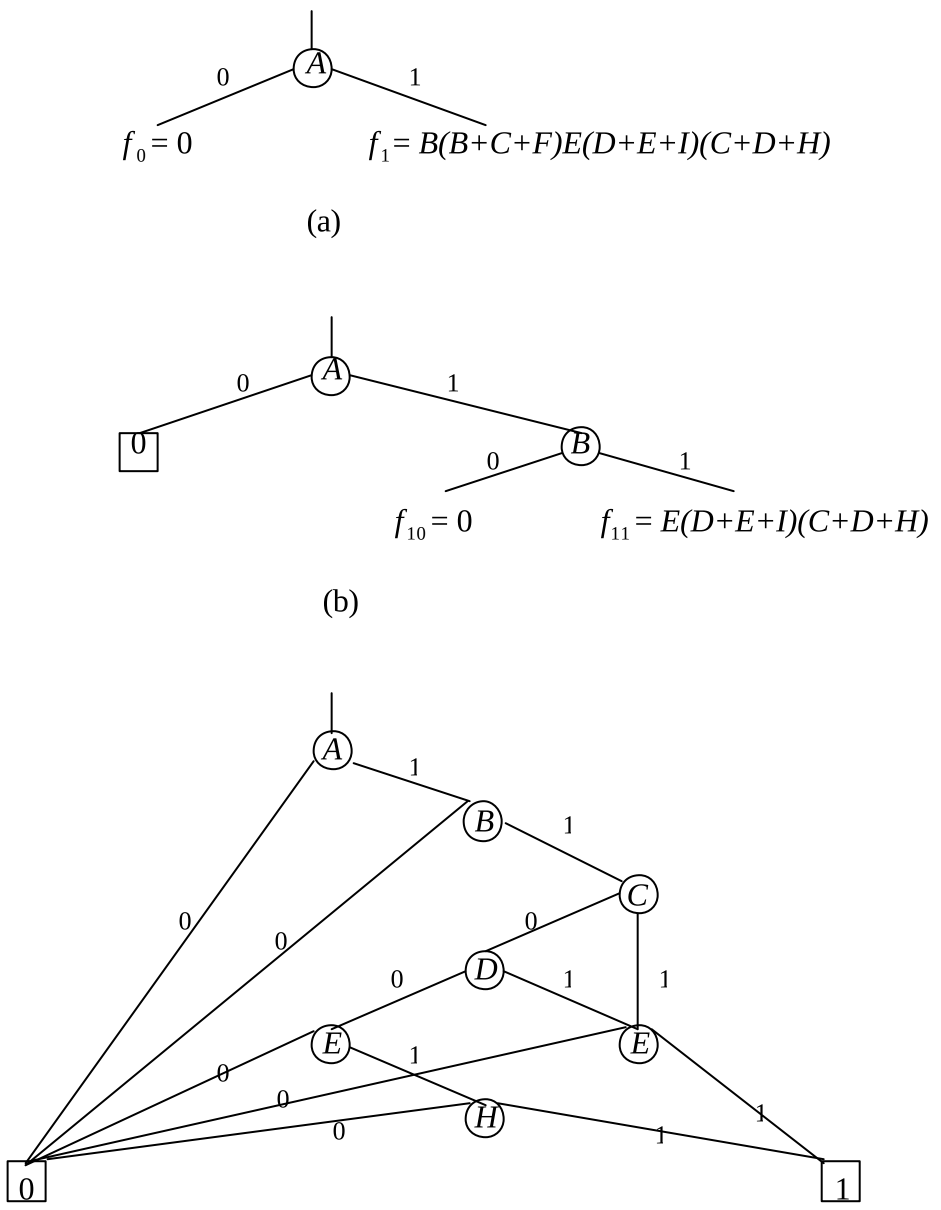


Fig. 3.8. Example BDD for cover table.

and, the output portion, which specifies which of the output functions include the input portion cube. Each symbol of the input portion corresponds to one of the variables and is 0, 1, or -. Each symbol of the output portion corresponds to one of the output functions and either 0 or 1 depending on whether the corresponding input portion cube is not included in the output function or is included in the output function. A multi-output function represented in this way is shown in Fig. 3.9.

w	x	y	z	f_1	f_2
0	0	0	0	1	0
0	1	0	1	1	1
0	1	1	1	1	1
1	1	1	1	1	1
1	1	1	0	1	1
1	0	0	0	1	0
1	0	0	1	0	1

Fig. 3.9. Multi-output function table and symbol.

<u>Pairing Generation and Odd/Even Covering Checking</u>

The first step in pairing the multi-output function cubes is to compare each pair of cubes to determine whether or not the input portions differ by only one literal. If two cubes that satisfy this condition are found, a new cube is formed. The input portion of the new cube is formed in the same way as in the single-output technique. The output portion of the new cube will have 1’s in all positions in which both of the original cubes have 1’s in the same position, and 0’s in the remaining positions. The reason behind this pairing rule is that the new cube corresponds to a product term that is included in those functions that include both the original products used in forming the product term. This is why there are 1’s in the output portion of new cube only where there are 1’s in both the output portion of the original cubes.

Just as in the single-output technique, it is necessary to check off some of the cubes, since they do not all correspond to prime implicants. The rule for multi-output function is that if a new cube is formed, the 1’s in all positions in which both of the original cubes have 1’s in the same position will change to don’t cares. The don’t care in the output portion does not have the same meaning as in the input portion. It means that the product term which is included in this output function is not a prime implicant and can be discarded after completion of the process. If the original cube in output portion is a don’t care, it can still be used to form a new cube with the other original cube. The cubes whose output portions still have 1’s will be added to the new list after completion of the process. Of course, a new cube with an all-0 output portion need not be written down, since it corresponds to a product term which is not included in any of the output functions. In the example as shown in Fig. 3.9, the original cube “0000|10” and “1000|10” can merged into the new cube “-000|10”. Then both of the original cubes will change to “0000|-0” and “1000|-0” after completion of the pairing process. As an example, the original cube list is shown in Fig. 3.10a, and the generated cubes are shown in Fig. 3.10b. In Fig. 3.10a, the original cube “1001|01” in output portion still has a 1 left. Thus, it will be added into the new list as shown in Fig. 3.10c.

w	x	y	z	f_1	f_2
0	0	0	0	-	0
0	1	0	1	-	-
0	1	1	1	-	-
1	1	1	1	-	-
1	1	1	0	-	-
1	0	0	0	-	0
1	0	0	1	0	1

(a)

w	x	y	z	f_1	f_2
-	0	0	0	1	0
0	1	-	1	1	1
-	1	1	1	1	1
1	1	1	-	1	1

(b)

w	x	y	z	f_1	f_2
-	0	0	0	1	0
0	1	-	1	1	1
-	1	1	1	1	1
1	1	1	-	1	1
1	**0**	**0**	**1**	**0**	**1**

(c)

Fig. 3.10. Pairing generation for multi-output function.

The second part of first step for multi-output functions involves the odd/even cover checking. The original minterm cannot be automatically deleted unless it is covered an even number of times for all the output functions. Otherwise, a new cube needs to be added into the list. The input portion of this new cube is the same as the original minterm and the output portion of the new cube is differs slightly from the original minterm. If the minterm is covered an even number of times in an output function, the output portion of this new cube will have 1 in the position corresponding to this output function. This technique is explained with an example. The original cube “0000|10” is only covered by cube “-000|10”. Thus, we do not have to put a new cube in the list. The original cube “0111|11” is covered by both cube “01-1|11” and “-111|11”. So, we have to put a new cube “0111|11” into the list. The output portion of this new cube has 1’s in all the positions because the minterm is covered twice by all the output function. In our example, after completion of this process the cubes generated are shown in Fig. 3.11.

w	x	y	z	f_1	f_2	
-	0	0	0	1	0	*A*
0	1	-	1	1	1	*B*
-	1	1	1	1	1	*C*
1	1	1	-	1	1	*D*
1	0	0	1	0	1	*E*
0	1	1	1	1	1	*F*
1	1	1	1	1	1	*G*

Fig. 3.11. Odd/Even cover checking for multi-output function.

<u>Cover Table Solving</u>

The cubes in Fig. 3.11 have been marked as *A, B, C, D, E, F, and G.* A cover table can be built as shown in Fig. 3.12. There must be a column of this table for each minterm of each of the output functions. The table is partitioned into sets of columns so that all the minterms corresponding to one of the output functions are represented by a set of adjacent columns. For example, In Fig. 3.12 there are two columns labeled *m5*, one for f_1

and one for f_2. Each row of the table corresponds to one of the multiple-output prime implicants. These are also partitioned into sets of rows by listing first the rows that correspond to prime implicants of f_1, then those for f_2, …, those for $f_1 * f_2$, etc. The cover table is solved in the same way as in the single-output technique. By solving the cover table, we get $f = \{A, B, D, E\}$. The results of this multiple output circuit are shown in Fig. 3.13.

		f_1						f_2				
		m0	*m5*	*m7*	*m8*	*m14*	*m15*	*m5*	*m7*	*m9*	*m14*	*m15*
f_1	*A*	x			x							
f_2	*E*									x		
$f_1 * f_2$	*B*		x	x				x	x			
	C			x			x		x			x
	D					x	x				x	x
	F			x					x			
	G						x					x

Fig. 3.12. Cover table for multi-output function.

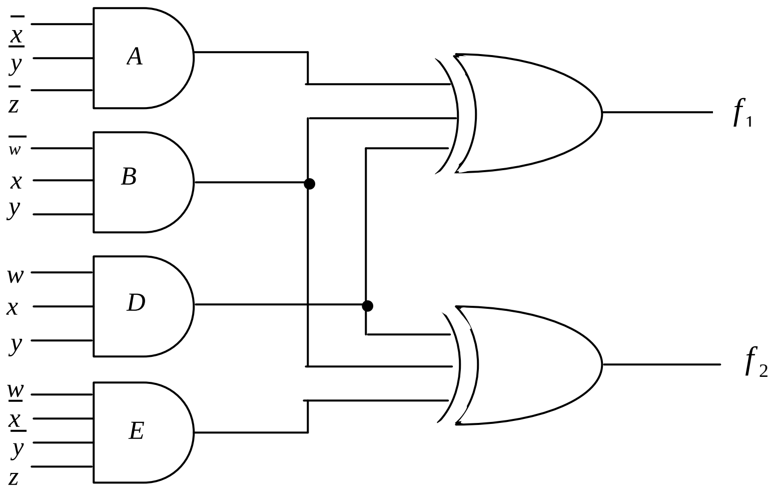


Fig. 3.13. ESOP circuit resulting from cover solution.

CHAPTER IV

IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this chapter, the implementation of this technique is described and experimental results are presented.

Implementation

As we introduced in chapter 4, the tabulation method requires an excessive amount of memory and computation time if the input cube list is large. Thus, care must be taken in order to reduce the memory and computation time when this technique is implemented.

Cube List Sorting

The first part of step 1 in the technique is pairing generation. Each pair of *(k-1)*-cubes is checked to see if they differ by only one literal. For each cube, consider the straightforward implementation algorithm that simply starts searching at the beginning of a cube list and searches until either the cube is found or the end of the list is reached. In the worse case, this implementation requires $O(N^2)$ cube comparisons. In terms of function inputs, this bound is exponential, $O(2^n)$. If cube list is large, the program will spend a lot of CPU time searching for pairs to merge.

When the values of a list are in a sorted order, there are better searching methods than the straightforward algorithm just described. For example, when a name is looked up in a phone book, it is common not to start in the beginning and scan through until the name is found. Instead, the fact that the names are listed in sorted order is used to jump quickly to the right page before scanning. Thus, the ordering of the list can be taken advantage of and a binary search may be used. Until the list being considered is empty or the cube is found, the technique considers the cube at the midpoint in the list.

1) If this is the cube that is being sought, the task is finished.
2) If the cube being looking for is less than the middle cube, eliminate the middle cube and all cubes greater.
3) If the cube being sought is greater than the middle cube, eliminate the middle cube and all cubes smaller.

Therefore, if the cube list is in sorted order instead of unsorted order, a binary search method can be used. Consider a pair of cubes that can be merged to generate a new cube, the pair should have following two properties:

1) The pair of cubes must have the same cube number. For example, 1-cube is generated from pair of 0-cubes, and 2-cube is generated from pair of 1-cubes.
2) The number of 1's is differed by one. If two cubes can be paired, one cube must have 1, and the other cube must have 0 in the corresponding position, then the rest of the literals in the corresponding position must be the same. For example, cube "1000" and "1001" can be paired, the number of 1's in cube "1000" is one, and two in cube "1001".

Thus, the cube list can be sorted that is based on the two pairing properties discussed above before searching for pairs. In the cube list, the number of cubes are sorted in sequential order, and the number of 1's are sorted in sequential order for those cubes have the same number of cubes. Consider the unsorted cube list shown in Fig. 4.1a, and the sorted cube list shown in Fig. 4.1b. There are a total of ten cubes in the lists. In Fig. 4.1a, pairs for the cube "0000" are being sought. It is necessary to compare cube "0000" with all nine of the other cubes in the list. However, in Fig. 4.1b, it is only necessary to compare cube "0000" with the cubes in the 0 and 1 subset of cubes. There are two cubes in this section. Thus, we only have to compare cube "0000" with two cubes instead of nine cubes after cube list is sorted.

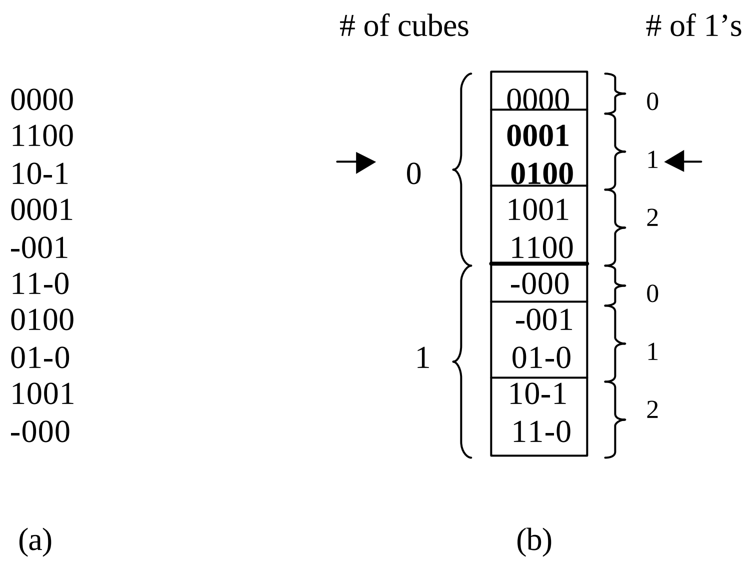


Fig. 4.1. Unsorted cube list versus sorted cube list.

The program is implemented using this method. The experimental results show that the program runs much faster than before by using this technique.

Odd/Even Cover Checking

The second part of step1 and step2 involves the odd/even cover checking. Each minterm is checked to see whether they are covered an odd number of times or even number of times. When this step is implemented in a straightforward manner, it is necessary to check each minterm covered by the cubes in the cube list one by one. If the cubes in the cube list cover too many minterms, a huge computation time will result.

The basic idea for speeding up this process is to check if a cube does not have intersection with any other cube in the cube list. In this case, the minterms covered by this cube are covered an odd number of times. If a cube has intersection with another cube in the list, the minterms covered by the both of these two cubes are covered an even number of times. Thus, when the odd/even cover process is implemented, it is not necessary to check each minterm one by one. Instead, it is only required to check whether the cube has intersection with another cube. As an example, consider the function as shown in Fig. 4.2. Cube *C* is the intersection of cube *A* and *B*. Cube *A* and *B* cover ten minterms in total. If each minterm is checked one by one, it is necessary to check ten times and put two minterms “0101”, “0111” in the list. However, if only the intersection of two cubes is checked, the technique simply checks once and puts cube *C* in the final list.

Thus, if the program is implemented using this method, it has two advantages. First, it saves computation time by avoiding a check over each minterm. Second, it saves memory by avoiding storage of the minterms.

0---	*A*
-1-1	*B*

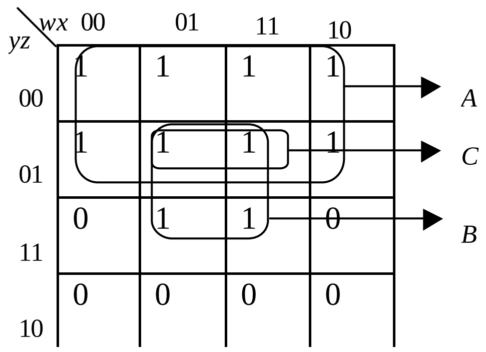


Fig.4.2. Odd/Even cover checking.

Positional Cube Notation (PCN)

This technique has been implemented based on the use of cube lists. The initial input is a list of covering cubes in PLA (ESPRESSO) format. Each input cube consists of symbols. Each symbol of the cube corresponds to one of the variables and is 0, 1, -, ∅. ∅ means is a null-literal and indicates a cube is void and should be deleted from the set. The symbol, “–“ indicates a “Don’t Care”. 1 indicates that the literal is an uncomplemented variable and 0 indicates that the literal is a complemented variable.

Positional Cube Notation (PCN) is used to represent each symbol. PCN encodes each symbol as a 2-bit field:

∅	:	00
1	:	01
0	:	10
-	:	11

As an example, consider the function, $f = \bar{a}\,\bar{d} + \bar{a}b + a\bar{b} + a\bar{c}d$. The implicant table in PCN is shown as follows:

10	11	11	10	$\bar{a}\,\bar{d}$ (0--0)
10	01	11	11	$\bar{a}b$ (01--)
01	10	11	11	$a\bar{b}$ (10--)
01	11	10	01	$a\bar{c}d$ (1-01)

PCN representation enhances the efficiency of manipulation although it needs more memory to store the cubes. For example, cube intersection is very easy since it is simply the bit-wise product. Consider the intersection of $\bar{a}\,\bar{d}$ and $\bar{a}b$ shown as the following PCN operation:

$$\begin{array}{cccc}
10 & 11 & 11 & 10 \\
10 & 01 & 11 & 11 \\
\hline
10 & 01 & 11 & 10
\end{array} \longrightarrow \bar{a}b\bar{d} \text{ (01-0)}$$

Experimental Results

This program was implemented and run using standard benchmark circuits and the results were compared to the ESOP minimizers EXMIN2 and MINT. In TABLEs 2, 3, and 4, *products* means the number of products in the AND-EXOR realization. *min* means the implemented program, TABESOP, uses the minimum minterm count approach. *max* means TABESOP uses maximum minterm count approach.

TABLE 2 compares TABESOP with EXMIN2 and MINT. TABESOP uses the two heuristic approaches and is compared with EXMIN2 and MINT. EXMIN2 and MINT generate better results. TABLE 3 compares TABESOP with DJ. DJ is the program that transforms the initial expression into a disjoint sum-of-products expression (DSOP). Before MINT and EXMIN2 are invoked, the initial input must be transformed into an ESOP. Usually a DSOP is used by running DJ. Comparing with DJ, TABESOP generates better results. In TABLE 4, the initial input (SOP form) for MINT is transformed into an ESOP form by running DJ and TABESOP. Compared with DJ, MINT generates roughly equal results by using TABESOP. TABLE 5 compares cover table size with BDD size. From TABLE 5, we can see that BDD approach for representing the cover table requires less memory than cube list approach.

TABLE 2

Comparison with EXMIN2 and MINT

Circuit	EXMIN2 (products)	MINT (products)	TABESOP (no rules)	
			min (products)	max (products)
5xp1	34	32	126	126
9sym	53	51	223	219
b12	28	28	253	255
clip	68	64	403	413
f51m	32	31	147	147
in7	35	35	470	624
rd53	15	15	34	34
rd73	42	36	149	149
rd84	59	55	1459	414
sao2	29	29	220	220
vg2	184	184	816	816

TABLE 3

Comparison with DJ (transform SOP into DSOP)

Circuit	SOP	DJ (DSOP)			TABESOP					
					min			max		
	products	products	literals	time (s)	products	literals	time (s)	products	literals	times (s)
b10	140	204	5611	0.06	158	1480	0.72	162	1528	0.70
dc2	58	85	548	0.03	58	326	0.08	58	326	0.10
co14	47	14	196	0.02	14	196	0.01	14	196	0.01
risc	74	109	585	0.02	42	175	0.21	42	172	0.19
bw	87	269	1092	0.03	114	860	0.19	114	860	0.20
inc	34	132	711	0.10	57	276	0.10	57	276	0.09
misex1	32	38	163	0.02	25	105	0.02	25	105	0.03
mux	36	52	412	0.04	16	96	0.16	16	96	0.09
x2	35	91	515	0.03	82	330	0.06	82	330	0.08
log8mod	47	99	562	0.03	60	298	0.10	60	298	0.08
b11	74	143	735	0.02	45	184	0.29	45	184	0.27
cm85a	48	112	896	0.02	78	588	0.09	78	588	0.10
duke2	87	266	2613	0.03	150	1341	0.61	151	1347	0.70
sex	23	45	184	0.01	40	158	0.04	40	158	0.04

TABLE 4

MINT Using DJ and TABESOP

Circuit	Mint Using DJ			Mint Using TABESOP					
				min			max		
	products	literals	time (s)	products	literals	time (s)	products	literals	times (s)
b10	85	755	20.07	91	839	27.77	92	847	21.79
dc2	32	151	0.85	32	151	0.96	32	151	0.88
co14	14	182	0.02	14	182	0.01	14	182	0.01
risc	27	112	0.63	27	112	0.56	27	112	0.63
bw	22	110	0.31	22	110	0.22	22	110	0.28
inc	29	134	0.80	29	138	0.87	29	138	1.00
misex1	12	47	0.15	12	47	0.14	12	47	0.15
mux	16	16	0.48	16	96	0.46	16	96	0.44
x2	15	61	0.26	15	61	0.25	15	61	0.24
log8mod	27	110	1.00	27	110	0.93	29	122	0.88
b11	26	117	0.59	26	110	0.61	26	110	0.56
cm85a	48	270	2.61	48	270	2.51	48	270	2.48
duke2	78	684	32.81	80	703	31.42	78	694	38.35
sex	23	89	0.53	22	83	. 054	23	89	0.53

TABLE 5

Cover Table Size Comparison with BDD Size

Circuit	Cover Table (bytes)	BDD (bytes)	% Difference
add6	45880	4746	867%
b12	11843	253	4581%
clip	1813	403	350%
dist	1955	494	296%
intb	40296	4577	780%
rd84	16983	414	4002%
cm162a	1320	162	715%
cm163a	1066	130	720%
exp	1428	283	405%

CHAPTER V

CONCLUSION

In this thesis, a heuristic ESOP minimization technique based on the primitive operation of odd/even cube covering is described, and formulated as a minimization problem of ESOPs for multi-output functions. A minimization algorithm called TABESOP, which is derived by modifying the well-known tabulation method used for SOP minimization is developed.

The program was implemented in $C++$, and results have been compared to other ESOP heuristic minimizers. Methods for implementing this technique using cube lists and BDDs are described in chapter 3. Two approaches were used to solve the cover table. The first approach is based on minimum minterm counts, and the second approach is based on maximum minterm counts. The reasons these two approaches are used to solve the cover table are based on the following facts. Each time a new cube is added to the final list, the odd/even counting process must be invoked. The minimum minterm count method usually results in adding fewer minterms to cover table during odd/even counting. The maximum minterm count method can cause more minterms to be added to cover table, but allows more rows to be deleted. Our experimental results show that both methods can get better results in different cases.

Methods for implementing this program were also described. Performance enhancements in terms of speed and required memory were described by using more sophisticated searching and merging algorithms and by using BDD and PCN data structures.

Future extensions are research into techniques for solving the binate form of Petrick’s equation based on a BDD representation of the cover table, and the investigation of the use of other methods for the representation of the product term list. Also, incorporation of more rules after initial solution of the cover table will enhance the results.

REFERENCES CITED

[1] S. Devadas, A. Ghosh, and K. Keutzer. Logic Synthesis. McGraw-Hill, Inc., 1994.

[2] M. Karnaugh. The map method for synthesis of combinational logic circuits. Transactions of A.I.E.E., 72(pt. 1):593-599, Nov. 1953.

[3] E.J. McCluskey. Minimization of Boolean functions. Bell System Technical Jour., 35, 1956.

[4] W. Quine. A way to simplify truth functions. Amer. Math. Monthly, 62:627-631, 1955.

[5] E. W. Veitch. A chart method for simplifying truth functions. Proc. Assoc. Comp. Mach., May:127-133, 1952.

[6] U. Rollwage. The complexity of mod-2 sum PLA's for symmetric functions. IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design, pages 6-12, 1993.

[7] T. Sasao and Ph. Besslich. On the complexity of mod-2 sum PLAs. IEEE Trans. on Comp., 39:262-266, 1990.

[8] T. Sasao. Switching Theory For Logic Synthesis. Kluwer Academic Publisher, 1999.

[9] M.R. Garey and D.S. Johnson. Computers and Intractability – A Guide to NP-Completeness. Freemann, San Francisco, 1979.

[10] O. Coudert and J. C. Madre and H. Fraisse, "A New Viewpoint on Two-Level Logic Minimization," DAC, pp. 625-630, 1992.

[11] P. McGeer, J. Sanghavi, R. K. Brayton and A. Sangiovanni-Vincentelli, "Espresso-Signature: A New Exact Minimizer for Logic Functions," DAC, pp. 618-624, 1992.

[12] S.J. Hong, R. G. Cain, and D. L. Ostapko. Mini: A heuristic approach for logic minimization. IBM J. Res. And Develop., 18:433-458, 1974.

[13] R. Rudell and A. Sangiovanni-Vincentelli, “Espresso-MV: Algorithms for multiple-valued Logic Minimization,” in Proc. IEEE Custom Integrated Circuits Conf., 1985, PP. 230-234.

[14] G. De Micheli. Synthesis and Optimization of Digital Circuits. McGraw-Hill, Inc., 1994.

[15] G. Hachtel and F. Somenzi. Logic Sythesis and Verification Algorithms. Kluwer Academic Publisher, 1996.

[16] T. Sasao. EXMIN2: A simplification algorithm for Exclusive-OR-Sum-of products expressions for multiple-valued-input two-valued-output functions. IEEE Trans. on CAD, 12:621-632, 1993.

[17] Pradhan, D.K.: “Universal test sets for multiple fault detection in AND-EXOR arrays”, IEEE Trans. On Comput., Vol. C-27, No.2., pp.181-187, 1978.

[18] M. Helliwell and M. Perkowski. A fast algorithm to minimize multi-output mixed-polarity generalized reed-muller forms. In Design Automation Conf., pages 427-432, 1988.

[19] R.K. Brayton, G.D. Hachtel, C. McMullen, and A.L. Sangiovanni-Vincentelli. Logic Minimization Algorithms for VLSI Synthesis. Kluwer Academic Publishers, 1984.

[20] R.E. Bryant. Graph – based algorithms for Boolean function manipulation. IEEE Trans. On Comp., 35(8):677-691, 1986.

[21] E.L. Dagless T. Kozlowski and J.M. Saul. An enhanced algorithm for the minimization of exclusive-OR sum of products for incompletely specified functions. In Proceedings of the IEEE International Conference on Computer Design (ICCD95), pages 244-249. IEEE Computer Society Press, October 1995.

[22] S. R. Petrick, “A direct determination of the irredundant forms of Boolean function from the set of prime implicants,” Tech. Rept. AFCRC-TR-56-110, Air Force Cambridge Res. Center, Cambridge, MA. April 1956.