



Minimization methods for cellular logic arrays
by Harold Gregory Schmitz

A thesis submitted to the Graduate Faculty in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY in Electrical Engineering
Montana State University
© Copyright by Harold Gregory Schmitz (1969)

Abstract:

This thesis is concerned with the development of minimization techniques for the efficient synthesis of cellular logic networks.

The main results can be summarized as follows: First, an algorithm is presented which obtains a realization for an arbitrary switching function using a minimal size two-dimensional cellular logic array in which each cell is capable of producing all two-input one-output unate functions and in which the interconnection structure is constrained to be "nearest neighbor." Also, the number of equivalence classes of unate cascade realizable functions is determined, and a necessary condition and improved test for determining unate cascade realizability is given. Second, a solution to a classical minimization problem involving the Reed-Muller canonic forms is provided and minimization techniques for two-dimensional cellular arrays based on the Reed Muller forms are developed. Third, an essential step in the cellular array minimization algorithms is formulated as a graph theoretic problem whose solution requires finding the cliques of a linear nondirected graph. For this, a new algorithm for detecting the cliques of a linear nondirected graph is developed.

MINIMIZATION METHODS FOR CELLULAR LOGIC ARRAYS

by

HAROLD GREGORY SCHMITZ

A thesis submitted to the Graduate Faculty in partial
fulfillment of the requirements for the degree

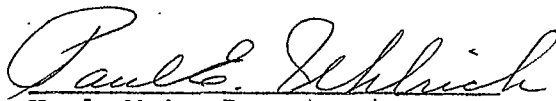
of

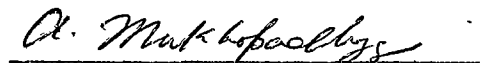
DOCTOR OF PHILOSOPHY

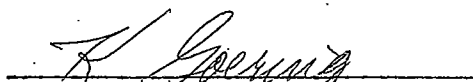
in

Electrical Engineering

Approved:


Head, Major Department


Chairman, Examining Committee


Graduate Dean

MONTANA STATE UNIVERSITY
Bozeman, Montana

December, 1969

ACKNOWLEDGMENT

The author wishes to express his gratitude to Dr. Amar Mukhopadhyay for his many suggestions and productive discussions during the course of this thesis research.

The financial support of the graduate study, furnished by National Defense Education Act Title IV Fellowship (No. 66-07474) and National Science Foundation Grant No. GJ-158, is also gratefully acknowledged.

The author particularly wishes to thank his wife, Jacquie, and children, Jennifer and Kurt, for their moral support.

TABLE OF CONTENTS

Chapter 1:	INTRODUCTION AND REVIEW OF CELLULAR RESEARCH . . .	1
1.1	Introduction	2
1.2	Review of Previous Work and Delimitation . . .	4
1.3	Organization of Remaining Chapters	8
Chapter 2:	MINIMIZATION OF UNATE CELLULAR ARRAYS.	10
2.1	Introduction	11
2.2	Unate Concepts	11
2.2.1	Definition	11
2.2.2	Unate Cascades and Properties of Functions Realizable by Unate Cascades	12
2.2.3	Unate Cascade Realizable Equivalence Classes.	15
2.2.4	Unate Cascade Realizability Test . .	22
2.2.5	Two-Dimensional Arrays	28
2.3	Minimization Algorithm	31
2.3.1	Definitions.	31
2.3.2	Algorithm Steps.	33
2.4	Disadvantages of the Previous Algorithm.	36
2.5	Improved Minimization Algorithm.	37
2.5.1	Detection of the Basic Unate Components	37
2.5.2	Improved Algorithm Steps	43

2.6	Algorithm Extensions	49
2.6.1	Optional States.	49
2.6.2	Multiple Output Functions.	49
2.6.2.1	Introduction.	49
2.6.2.2	Determination of Common Unate Components.	50
2.7	Variations of Unate Cascade Results.	57
Chapter 3:	MINIMIZATION OF CELLULAR ARRAYS BASED ON REED-MULLER CANONIC FORMS	60
3.1	Introduction	61
3.2	Elementary Definitions and Canonical Forms	61
3.3	Minimization Problems.	68
3.3.1	The First Problem.	72
3.3.2	The Second Problem	89
3.3.3	Functions with Optional States	92
3.3.4	Multiple-Output Functions.	92
3.4	Minimal Two-Dimensional Arrays	93
3.5	Cascade Realizability Test for Functions in the Reed-Muller Forms.	95
3.6	Minimal Arrays with Preset Variable Orderings.	99
3.7	A Conjecture	106
Chapter 4:	CLIQUE DETECTION IN LINEAR NONDIRECTED GRAPHS.	107
4.1	Introduction	108
4.2	Definition	109
4.3	Algorithms	111

4.3.1	Algebraic Formulation.	111
4.3.2	Method of Harary-Ross.	116
4.3.3	Even's Method	121
4.4	Comparison of Methods.	125
Chapter 5:	SUMMARY AND RECOMMENDATIONS FOR FUTURE RESEARCH	127
5.1	Summary.	128
5.2	Suggestions for Future Research.	129
5.2.1	The Variable Ordering Problem.	129
5.2.2	Decomposition Approach	131
5.2.3	Iterative Decomposition.	135
APPENDIX		137
LITERATURE CITED		142

LIST OF TABLES

Table 2.1	Statistics on Cascade Realizable Functions. . .	21
Table 2.2	Inclusion Table for $f(w,x,y,z) = \sum (1,5,7,8,10,11,13,15)$	40
Table 2.3	Reduced Inclusion Table	41
Table 2.4	Basic Unate Components.	43
Table 2.5	Cover Table for $f = \sum (1,5,7,8,10,11,13,15)$. .	47
Table 2.6	Reduced Cover Table	47
Table 2.7	Implicants for f_1 , f_2 , and f_3	52
Table 2.8	Prime Unate Components for f_1 , f_2 , and f_3 . . .	55
Table 2.9	Multiple Output Cover Table	56

LIST OF FIGURES

Figure 1.1	Array of Two-Input One-Output Cells.	7
Figure 2.1	A One Dimensional Cascade of Unate Cells	12
Figure 2.2	Unate Cascade of AND-OR Cells	17
Figure 2.3	Unate Cascade Realizable Functions of Four Variables.	19
Figure 2.4	A Partial Realization for f	25
Figure 2.5	Unate Cascade Realizing $f = x_1x_3 + x_2'x_3 + x_4$	26
Figure 2.6	Two Dimensional Unate Cellular Array	29
Figure 2.7	A Partially Ordered Graph.	34
Figure 2.8	Compatibility Graphs for Table 2.3	42
Figure 2.9	Partially Ordered Graph for Unate Cascade Realizable Components.	46
Figure 2.10	Minimal Cascade Realization.	48
Figure 2.11	Multiple-Output Two-Dimensional Unate Cellular Array.	51
Figure 2.12	Inclusion Table for f_1 and f_3	54
Figure 2.13	Multiple-Output Realization.	58
Figure 3.1	Plot of $\vec{g} = (a_0, a_1, a_2, a_3)$ with all polarities of x_1 and x_2	73
Figure 3.2	The Polarity-Compatibility Graph for Example 3.1	86
Figure 3.3	Plot of the Polarity Functions $a_0(f)$, $a_1(f)$, $a_2(f)$, and $a_3(f)$	87
Figure 3.4	Plot of the Polarity Functions of Example 3.2. .	88
Figure 3.5	Cascade Realization for f	99

Figure 4.1	Linear Nondirected Graph and Associated Connection Matrix.	110
Figure 4.2	Connection Matrix for [<u>e</u> ,a,b,c,d].	115
Figure 5.1	Array with Permuter Network.	130
Figure 5.2	Doubly Complex Array	133

ABSTRACT

This thesis is concerned with the development of minimization techniques for the efficient synthesis of cellular logic networks.

The main results can be summarized as follows: First, an algorithm is presented which obtains a realization for an arbitrary switching function using a minimal size two-dimensional cellular logic array in which each cell is capable of producing all two-input one-output unate functions and in which the interconnection structure is constrained to be "nearest neighbor." Also, the number of equivalence classes of unate cascade realizable functions is determined, and a necessary condition and improved test for determining unate cascade realizability is given. Second, a solution to a classical minimization problem involving the Reed-Muller canonic forms is provided and minimization techniques for two-dimensional cellular arrays based on the Reed-Muller forms are developed. Third, an essential step in the cellular array minimization algorithms is formulated as a graph theoretic problem whose solution requires finding the cliques of a linear nondirected graph. For this, a new algorithm for detecting the cliques of a linear nondirected graph is developed.

Chapter 1

INTRODUCTION AND REVIEW
OF CELLULAR RESEARCH

1.1

Introduction

A cellular logic array is a one-, two-, or many-dimensional iterative arrangement of similar or identical combinational or sequential cells with a regular interconnection pattern on the cells. If the arrays are composed of simple cells (low complexity), they are referred to as microcellular. Arrays having more complex cells are called macrocellular.

Research in microcellular and macrocellular networks started about two decades ago. As early as 1952, studies in automata theory prompted research in the macrocellular area. Von Neumann⁽³⁷⁾ introduced the notion of a cellular space and discussed universal computing and constructing automata embedded in such a space.

A large part of the research in macrocellular networks has been motivated by the desire to build highly parallel computers. Holland's⁽¹⁸⁾ machine was a two-dimensional array of iterative modules. He envisioned it as a universal computer capable of simultaneously executing an arbitrary number of subprograms. Modified Holland machines were subsequently considered by Comfort⁽⁵⁾ and Gonzales.⁽¹²⁾ At the same time Holland was envisioning his iterative circuit computer, Unger⁽³⁶⁾ independently proposed a special purpose modular computer oriented to solve problems in which data is arranged in a spatial form.

More recently, investigations by Slotnick, Borck, and McReynolds (35) led to the SOLOMON and ILLIAC IV parallel processing computers.

Concurrent with the development of macrocellular arrays, there were other areas of development that would now be classified as microcellular in nature. The first of these areas might be termed contact arrays. The crossbar switch of Scudder and Reynolds (33) exhibited cellularity. Each "cell" of the crossbar switch consisted of several contacts and a mechanical latching arrangement. By operating magnets in the desired row and column in the proper sequence, it was possible to activate and hold various cells in the crossbar switch.

Diode arrays also led to the development of modern microcellular techniques. Brown (3) appears to have been the first to suggest the diode array which had an interconnection structure that consisted of a set of horizontal and vertical busses.

Memory arrays using shift registers were proposed by Aiken in about 1948. Shortly after, Forrester (11) and Rajchman (31) proposed the magnetic-core storage which was also microcellular in nature.

However, the early areas of microcellular development received only limited attention because of what were then technological limitations. It is only with the advent of integrated circuit and batch-fabrication technologies that microcellular techniques have taken on a

new importance. With the progress now being made in the field of large scale integration (LSI) technology, it is reasonably certain that the future generations of computers will employ cellular arrays in their completely integrated, highly parallel and sophisticated subsystems. The various advantages of cellular arrays in LSI are high packing density, high reliability, high manufacturing yield, flexibility in performance, and ease in error diagnosis.

To effectively utilize cellular arrays in LSI, logical designers must have efficient synthesis techniques at their disposal. The purpose of this research is the development of minimization algorithms for the efficient synthesis of cellular logic networks.

1.2 Review of Previous Work and Delimitation

Many cellular structures have been proposed for synthesizing combinational switching circuits. Maitra⁽²¹⁾ has shown that cascaded networks composed of completely flexible two-input one-output logical cells may be used to realize a certain class of switching functions. He also gives a condition for realizing arbitrary switching functions by such networks.

Minnick's cutpoint⁽²⁶⁾ cellular array was a two-dimensional rectangular arrangement of cells, each of which had binary inputs on the top and left edges and outputs on the bottom and right edges. Each cell

was connected with neighboring cells and specialized to produce a subset of the possible sixteen functions by a set of binary constants called "cutpoints". He showed that a cascade of cutpoint cells was capable of producing all the functions of the Maitra cascade. He also showed that a two-dimensional array of cutpoint cells could be arranged in such a way that an arbitrary function of n variables was produced. His synthesis technique relied on the ingenuity of the designer, however, and the realizations obtained did not necessarily have a minimum number of cells.

The cobweb array,⁽²⁵⁾ also introduced by Minnick, was a modification of the cutpoint array made by complicating the cell-interconnection structure. Improved realizations were possible with the cobweb array but the task of obtaining a realization for a particular function using a minimal number of cobweb cells was still up to the ingenuity of the designer.

Short⁽³⁴⁾ and Yoeli⁽³⁸⁾ studied two rail cascades. These cascades are functionally complete, i.e., any arbitrary function can be realized within the extent of a single cascade. However, the general two-output three variable cell of the two rail cascade is significantly more complex than the cell used in the arrays already described. Two rail realizations based on the Reed expansion may be obtained for an arbitrary function. However, this expansion does not always yield an

economical realization. Indeed, only when the number of variables is one or two is the bound firm on the number of cells needed to realize a particular function.

All of the cellular arrays mentioned so far have at least one drawback. Given a particular switching function, there exist no algorithms to find an optimal design of the array to realize the function. Perhaps this limited success in the development of optimal cellular arrays lies in the problem of mathematically modeling these arrays. Several design parameters are involved which makes modeling difficult. Synthesis algorithms must recognize the trade-offs among such design parameters as the size of the array, the logical complexity of individual cells, the interconnection structure, and the speed of computation.

The cobweb array illustrates the fact that the total array size for realizing any arbitrary function can be reduced by increasing the logical complexity of the cells and by allowing interconnection among non-adjacent cells. Research on the (n,d,k) problem⁽⁷⁾⁽⁸⁾ has produced some analytical results concerning the interdependence among some of the purely topological parameters of cellular arrays such as the total number of connections incident on a cell, maximum and average inter-cell distance, and the size and planarity of the array. When both logical and topological parameters are considered, mathematical modeling becomes very vague.

Mathematical modeling of cellular arrays is a problem which needs further study. However, it should be realized that the urge to attain excellence in modeling should not be carried so far as to obscure the necessity of developing practical design algorithms. Therefore, in this thesis we will start with a particular two-dimensional array whose interconnection structure is constrained as in Figure 1.1.

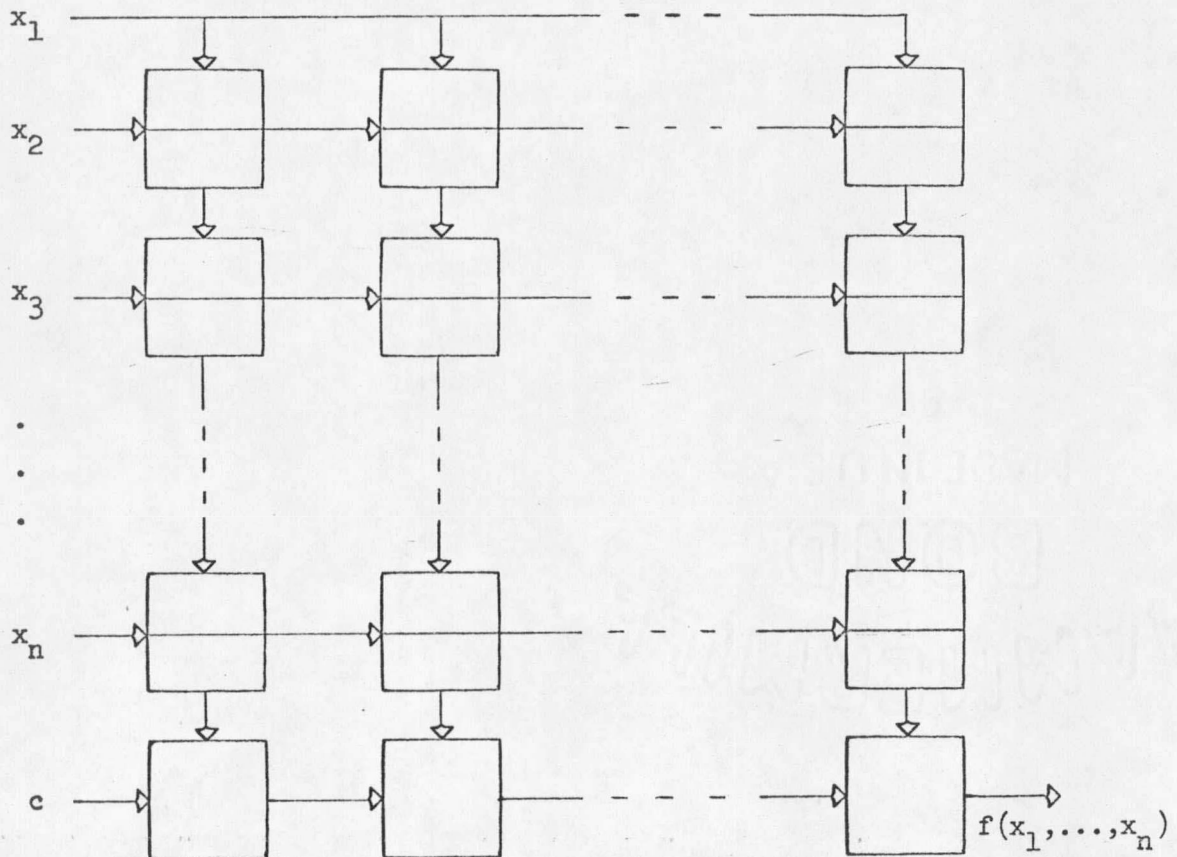


Figure 1.1 Array of Two-Input One-Output Cells.

Each cell produces an arbitrary two-input one-output combinational function. Input variables $x_1, x_2, \dots, x_n$ are bussed horizontally and with the exception of the bottom row, vertical cascades of cells do not interact. The problem of synthesizing two-dimensional arrays in which the vertical cascades are allowed to interact is still open and seems to be very difficult. The bottom row of Figure 1.1 acts as a "collector" row and collects the functions produced in each cascade.

Cellular arrays whose general structure fits that of Figure 1.1 are considered in the following chapters. Minimization algorithms can usually be developed only after some constraints are placed on the allowable cell functions. These constraints are briefly mentioned in the next section and more fully discussed in Chapters 2 and 3.

1.3 Organization of Remaining Chapters

Chapter 2 develops minimization techniques for the case in which the cascade functions produced in Figure 1.1 are restricted to be unate. Also, the number of equivalence classes of unate cascade realizable functions is determined, a necessary condition for unate cascade realizability is established, and an improved test for determining unate cascade realizability is given.

In Chapter 3, the collector row of Figure 1.1 is set to produce the Exclusive Or or Logical Equivalence operation. Minimization tech-

niques for arrays of this nature are developed and a solution to a classical minimization problem involving the Reed-Muller canonic forms is presented.

The algorithms developed in Chapters 2 and 3 require the determination of the cliques of a linear non-directed graph. Chapter 4 develops a new clique detection algorithm and compares it with other algorithms which have been programmed for this purpose.

The concluding Chapter 5 summarizes all the results obtained and discusses particular areas which merit further study.

Chapter 2

MINIMIZATION OF UNATE CELLULAR ARRAYS

2.1 Introduction

The most general minimization problems as discussed in the introduction to this thesis are very complex. A more tractable problem is to consider particular array structures and develop minimization algorithms applicable to them. The particular array that will be considered in this chapter is a unate cellular array which is a one- or two-dimensional arrangement of two-input one-output combinational cells. Each cell is capable of being set to give any one of the fourteen unate functions of its two inputs; that is, excluding Exclusive-Or and Equivalence.

2.2 Unate Concepts

2.2.1 Definition

A switching function $f(x_1, x_2, \dots, x_n)$ is said to be unate if and only if $f(x_1, x_2, \dots, x_n)$ is representable as a disjunctive or conjunctive normal formula in which no literal x_i appears complemented and uncomplemented.

The following is a list of all unate functions of two variables, x and y : $0, 1, x, x', y, y', \text{AND}(xy), \text{NAND}(x' + y'), \text{OR}(x + y), \text{NOR}(x' y'), \text{IMP}(x + y', x' + y), \text{NIMP}(x' y, xy')$. The first six functions of this list are improper functions of two variables; that is, they are independent of some or all input variables.

2.2.2 Unate Cascades and Properties of Functions Realizable by Unate Cascades

A one-dimensional array of two-input one-output unate cells as shown in Figure 2.1 is called a unate cascade. The variable ordering for Figure 2.1 is $x_1, x_2, \dots, x_n$. Note that the ordering begins with the first cell of the cascade.

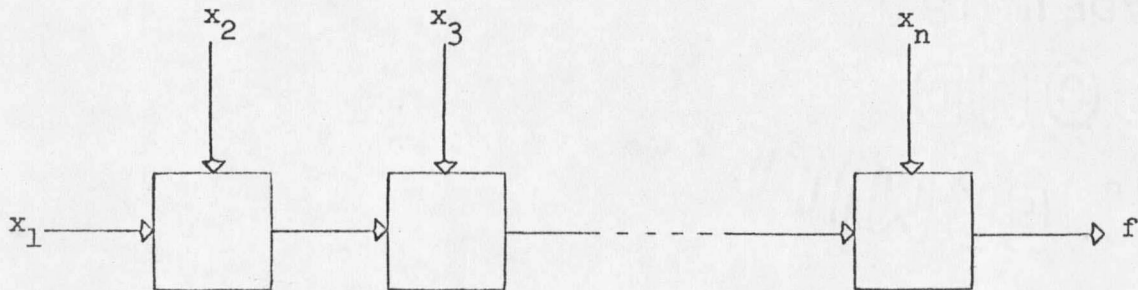


Figure 2.1 A One-Dimensional Cascade of Unate Cells.

This section presents some theorems on unate cascades and enumerates the classes of functions realizable by unate cascades.

Theorem 2.1: A function realizable by a unate cascade is unate.

Proof: See Reference 28.

The problem of determining the classes of unate cascade realizable functions can be made easier with the help of the following theorem.

Theorem 2.2: Consider a unate cascade of m cells with input variables $x_1, x_2, \dots, x_n$, where $m \geq n-1$. Every function realizable by this cascade could be realized by a unate cascade of at most m cells in which the only cell functions allowed are AND and OR and the input variables $x_1, x_2, \dots, x_n$ are allowed to be complemented or uncomplemented.

Proof: The output of the unate cascade in Figure 2.1 with input variables $(x_1, x_2, \dots, x_n)$ can be expressed as

$$f = x_n * (x_{n-1} * \dots x_3 * (x_2 * x_1)) \dots$$

in which $*$ denotes any of the unate functions of two variables. The function $g_1(x_1, x_2) = x_2 * x_1$ produced within the innermost parentheses can be realized using only an OR or AND cell if complementation of variables x_1 and x_2 is allowed.

Next consider the function $g_2(x_1, x_2, x_3) = x_3 * g_1(x_1, x_2)$ produced within the next innermost parentheses where $g_1(x_1, x_2)$ is an AND or OR function of x_1 and x_2 . Now $g_2(x_1, x_2, x_3)$ can likewise be realized using only AND or OR cells if the input variables are allowed to be complemented. This is so because if the $*$ operator requires the complement of $g_1(x_1, x_2)$, De Morgan's Law⁽²²⁾ ensures a representation for $g_1'(x_1, x_2)$ using the same number of AND and OR operators as $g_1(x_1, x_2)$.

If this argument is applied to each input variable in the cascade, the function f can be realized using a cascade of at most m cells

in which the only cell functions used are AND and OR. Hence the theorem.

One can think of the unate cascade of AND and OR cells of Theorem 2.2 as a "model" of all unate cascades since it can produce any unate cascade realizable function. This "model" is helpful in proving later theorems.

If, in the cascade of Figure 2.1, each variable is allowed to appear as an input to any number of cells, the cascade is called a redundant cascade. If each variable is applied to no more than one cell, the cascade is called irredundant. The purpose of the next theorem is to show that there is no loss of generality, as far as the number of realizable functions are concerned, in considering only irredundant unate cascades. First, the following lemma is needed.

Lemma 2.1: In any cascade realization of a function f , if the j th cell is an AND or OR function with input x_i , then there need never be any cells driven by x_i or x_i' among the first $j-1$ cells in the cascade.

Proof: See Reference 27.

Theorem 2.3: Every function f realizable by a unate cascade with redundant inputs can be realized by a unate cascade with irredundant inputs.

Proof: Theorem 2.2 says that any unate cascade of m cells can be re-

presented by a cascade of at most m AND and OR cells. Beginning with the last cell, Lemma 2.1 can be applied to each cell of the cascade. If an input variable x_i is encountered at the j th cell such that x_i or x_i' appears as an input to any of the first $j-1$ cells, by Lemma 2.1, the first $j-1$ cells can be replaced with $j-1$ or less cells in which x_i and x_i' do not appear. All redundant inputs can be eliminated in this manner and the theorem is proved. (Note that the irredundant unate cascade must be shorter in length than the redundant cascade.)

As a result of Theorem 2.3, we will hereafter consider only irredundant cascades.

2.2.3 Unate Cascade Realizable Equivalence Classes

The most common means of grouping switching functions of the same number of variables is by equivalence classes. Let $f(x) = f(x_1, x_2, \dots, x_n)$ and $g(x) = g(x_1, x_2, \dots, x_n)$ be switching functions of n variables. If $g(x)$ can be obtained from $f(x)$ by a permutation or complementation (or both) of the variables $x_1, x_2, \dots, x_n$, then $g(x)$ is said to be equivalent to $f(x)$ and vice versa; (i.e., $f(x)$ and $g(x)$ are in the same equivalence class.)

Further, $f(x)$ and $g(x)$ are said to be in the same general class if $g(x)$ can be obtained from $f(x)$ by a permutation or complementation (or both) of the variables $x_1, x_2, \dots, x_n$, or if $g(x) = f'(x)$.

This section derives the number of equivalence classes and genera classes generated by unate cascades. Although only a small percent of the total number of equivalence classes will be realizable by unate cascades, it will be shown in Section 2.2.5 that any arbitrary switching function can be realized using a two-dimensional array of unate cells.

Theorem 2.4: The number of equivalence classes of n variable switching functions realizable by unate cascades is $2^{n-1} + 2$ and the number of genera classes is $2^{n-2} + 1$.

Proof: Since we are not concerned with complemented and uncomplemented literals when discussing equivalence classes, Theorem 2.2 can be used to construct all the possible combinations of implicants generated by a unate cascade.

For an n variable unate cascade, the maximum number of AND and OR cells needed to represent the original cascade is $n-1$. This would occur when each cell function of the original unate cascade was a proper function of its two inputs.

To see how the equivalence classes might be generated, except for the trivial classes - constant 0 and constant 1, consider a unate cascade of $n-1$ AND-OR cells with ordering $x_1, x_2, \dots, x_n$ as in Figure 2.2.

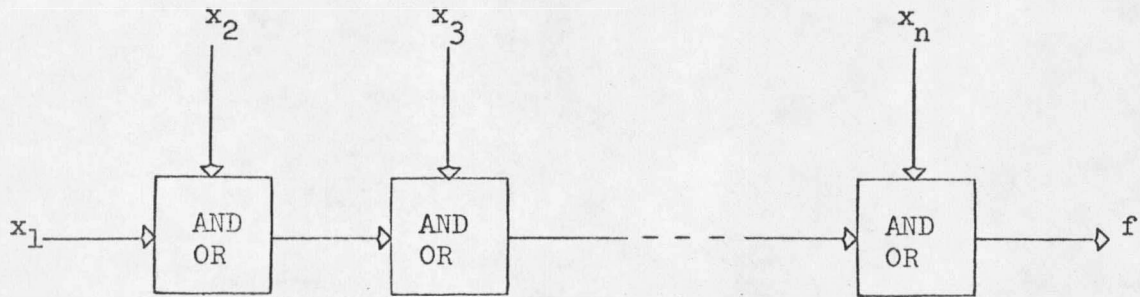


Figure 2.2 Unate Cascade of AND-OR Cells

The output of the first cell of the cascade will either be $x_1 + x_2$ (the sum of two $n-1$ cubes) or $x_1 x_2$ (one $n-2$ cube).^{*} Depending on the sequence of cell functions in the cascade, we will have as input to the i th cell a set of k -cubes, say $(k_1, k_2, \dots, k_t)$. If the function of the i th cell is OR, at the output of the i th cell we will have

^{*} Note there are 2^n n -bit binary vectors in an n -cube. A k -cube (or k -subcube) of an n -cube ($k \leq n$) is defined as a collection of any 2^k vectors which have exactly $(n-k)$ corresponding bits all the same. Each n -bit binary vector corresponds to a fundamental product of n literals. Each k -cube of the n -cube corresponds to a product of $(n-k)$ literals. For a four variable function $f(w, x, y, z)$, the implicant xyz is a 1-cube since it covers $2^1 = 2$ fundamental products. The implicant w is a 3-cube since it covers $2^3 = 8$ fundamental products.

a set of k -cubes $(k_1, k_2, \dots, k_t, n-1)$. If the function of the i th cell is AND, then at the output of the i th cell we will have as a set of k -cubes $(k_1-1, k_2-1, \dots, k_t-1)$. By considering all $n-1$ cells of the AND-OR cascade, at least 2^{n-1} equivalence classes can be generated. Adding the constant 0 and 1 classes gives a total of at least $2^{n-1} + 2$ equivalence classes. For example, all the unate cascade realizable functions of four variables (without complementation) with ordering x_1, x_2, x_3, x_4 are generated in Figure 2.3. For each realizable function there is a corresponding k -cube representation. For example, $x_1x_3 + x_2x_3 + x_4$ contains two 2-cubes and one 3-cube. The corresponding k -cube representation is $(2,2,3)$.

Each of the sequences in Figure 2.3 corresponds to at least one equivalence class of the unate cascade realizable functions. If we can show that each sequence corresponds to only one equivalence class, the number of equivalence classes will just be $2^{n-1} + 2$.

Since all prime implicants of a unate function are essential (see Reference 28), each function has a unique representation in terms of its prime implicants or corresponding k -cube. Suppose there are two cascade realizable functions f_1 and f_2 that have the same k -cube representation and belong to different equivalence classes. Now the k -cube sequences generated in Figure 2.3 are all distinct, so if f_1 and f_2 have the same k -cube representation, they must have been generated

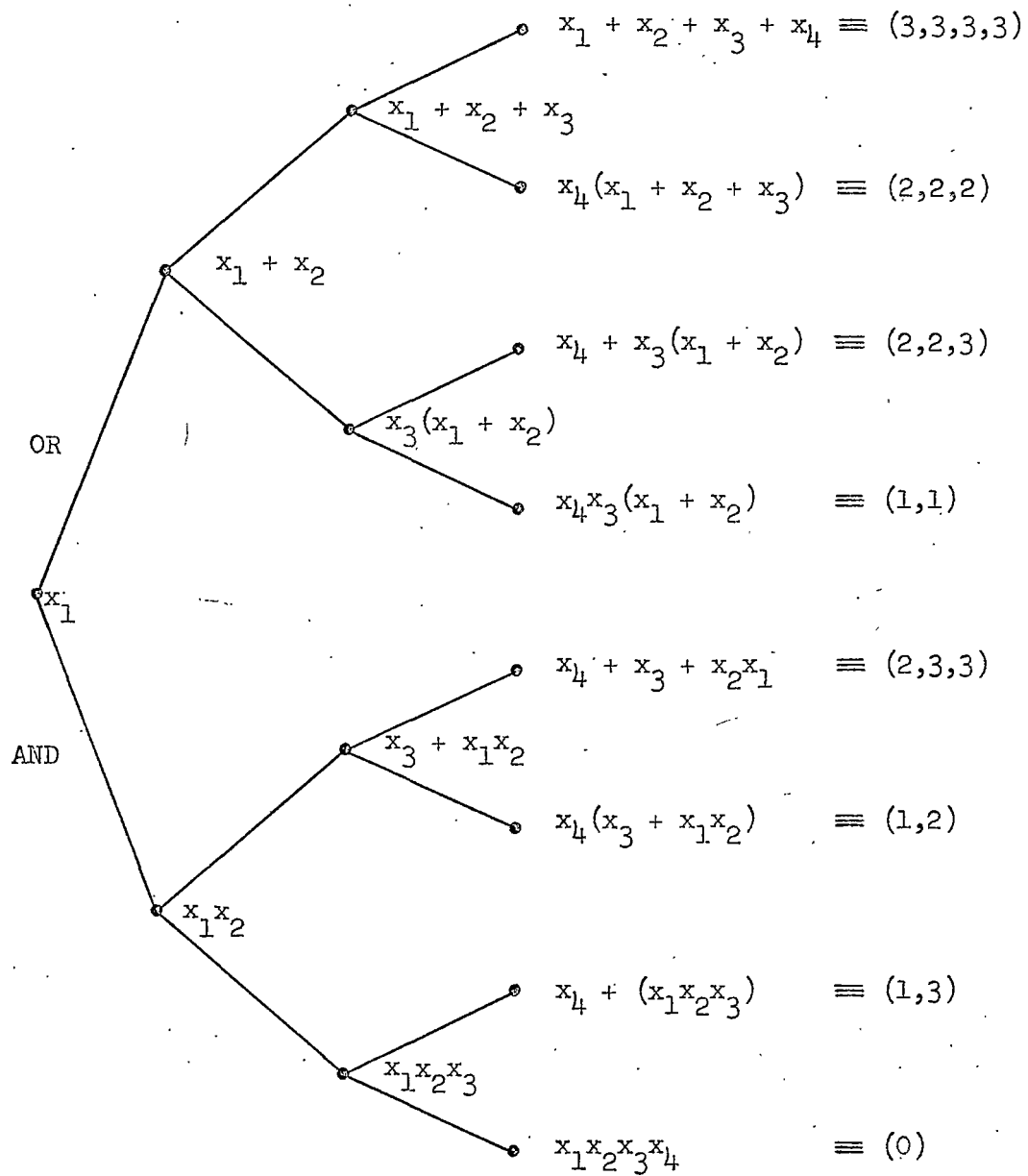


Figure 2.3 Unate Cascade Realizable Functions of Four Variables.

using exactly the same sequence of AND-OR cells. But if both functions were generated using the same sequence of cells, one must be obtainable from the other by a permutation or complementation (or both) of the input variables. Therefore, they must belong to the same equivalence class, and there are exactly $2^{n-1} + 2$ equivalence classes of n variable unate cascade realizable switching functions.

That the number of genera classes is just half the number of equivalence classes is obvious from looking at the symmetry of Figure 2.3. For each function generated, its complement (literals uncomplemented) also appears. Hence the number of genera classes of n variable unate cascade realizable switching functions is $2^{n-2} + 1$, and the theorem is proved.

The number of equivalence classes which are redundantly or irredundantly cascade realizable has been tabulated elsewhere.⁽²⁷⁾ In Table 2.1, the number of equivalence classes realizable using unate cascades and the number of equivalence classes realizable using any cascade of two-input one-output cells are shown.

Table 2.1 Statistics on Cascade Realizable Functions

No. Variables	2	3	4	5
Total No. Functions	16	256	65,536	$> 4 \times 10^9$
Total Equiv. Classes	6	22	402	1,228,158
Redundantly Cascade Realizable	6	19	135	$< 4,500$
Irredundantly Cascade Realizable	6	14	35	90
Unate Cascade Realizable	4	6	10	18

Note that the number of equivalence classes which are cascade realizable becomes diminishingly small no matter which type of cascade is used.

The following section gives a test procedure to determine whether a function is unate cascade realizable or not.

2.2.4

Unate Cascade Realizability Test

Given a switching function $f(x_1, x_2, \dots, x_n)$, we would like a test to determine whether f is realizable in a single unate cascade.

Theorem 2.4 of Section 2.2.3, and the subsequent construction of the unate cascade realizable equivalence classes in Figure 2.3 provide necessary conditions for a function to be realizable in a single unate cascade.

If an n variable function is realizable in a single unate cascade, its k -cube representation (in ascending order) must be identical to one of the k -cube representations for the $2^{n-1} + 2$ equivalence classes realizable by unate cascades.

For example, consider a four variable function $f(x_1, x_2, x_3, x_4)$. One way to test f for possible unate cascade realizability would be to find its k -cube representation and check to see if there was a matching k -cube representation in Figure 2.3.

A better way to check f for possible unate cascade realizability would be to use the k -cube representation and apply the steps used to generate the functions of Figure 2.3 in reverse order. If the k -cube representation for f could be decomposed, using a maximum of $n-1$ steps, to get a representation with two $n-1$ cubes (corresponding to $x_1 + x_2$) or

a representation with one $n-2$ cube (corresponding to x_1x_2), then f is possibly unate cascade realizable.

Provided f is possibly realizable, it can be further tested for definite realizability using a test given in Reference 28. This test and the necessary condition for realizability can be combined to give the following test and realization procedure for f :

Step 1: Obtain the prime implicants of the given function and check whether f is unate by forming the logical product of the prime implicants. The result must not equal zero if f is unate (proved in Section 2.5.1).

Step 2: The k -cube representation for f is listed with the k -cubes in ascending order.

Step 3: An attempt is made to remove an $n-1$ cube from the k -cube representation. If one is removable, remove it and go to Step 4. If not, go to Step 5.

Step 4: If the resulting k -cube representation is $(n-2)$ or $(n-1, n-1)$, the function is possibly realizable so go to Step 6. If the k -cube representation is otherwise, and the number of steps has not exceeded $n-1$, return to Step 3. If $n-1$ steps have been taken, the function is not realizable in a single unate cascade and testing stops.

Step 5: If less than $n-1$ steps have been taken, add 1 to each k -cube in the representation and go back to Step 3. If $n-1$ steps have been taken, the function is not realizable in a single unate cascade and testing stops.

Step 6: Check whether f can be expressed in one of the following forms:

$$A. \quad \Phi_1 = f = \dot{x}_{j_1} \dot{x}_{j_2} \dots \dot{x}_{j_k} \Phi_2 (X - x_{j_1} - x_{j_2} - \dots - x_{j_k})$$

$$B. \quad \Phi_1 = f = \dot{x}_{j_1} + \dot{x}_{j_2} + \dots + \dot{x}_{j_k} + \Phi_2 (X - x_{j_1} - x_{j_2} - \dots - x_{j_k})$$

where $X = (x_1, x_2, \dots, x_n)$, x_{j_i} ($1 \leq i \leq k$) is an element of X , $1 \leq k$ and Φ_2 is a function of X excluding $(x_{j_1}, x_{j_2}, \dots, x_{j_k})$.

Step 7: If f can be expressed either in form A or in form B, we have a partial realization of f as shown in Figure 2.4. The operation of the cell to which the input x_{j_i} is applied is as follows:

- a) If f is in form A, the cell performs the AND operation (xy) or the NIMP operation $(x'y)$ depending on whether

$$\dot{x}_{j_i} = x_{j_i} \text{ or } x'_{j_i}.$$

- b) If f is in form B, the cell performs an OR operation $(x+y)$ or an IMP operation $(x'+y)$ depending on whether $\dot{x}_{j_i} = x_{j_i}$ or x'_{j_i} .

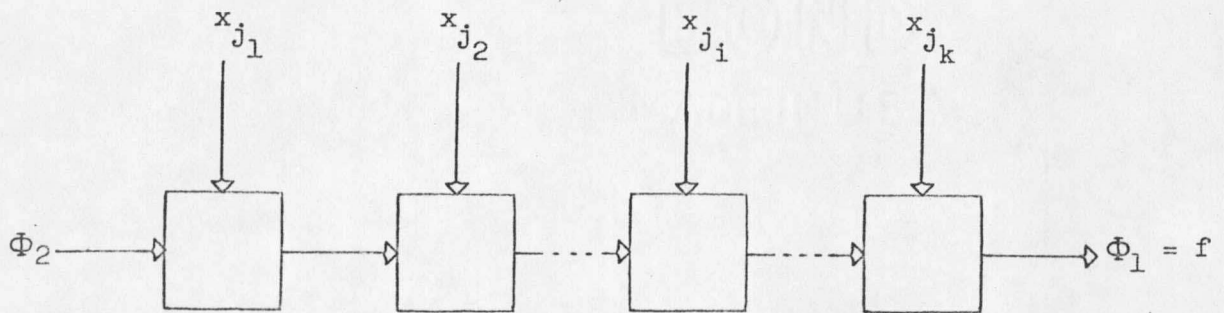


Figure 2.4 A Partial Realization for f .

Since f is symmetric with respect to $(x_{j_1}, x_{j_2}, \dots, x_{j_k})$, any permutation of the cells within the cascade of k cells is permitted and we have $k!$ partial realizations of f . If f cannot be expressed in either form A or B, then f is not realizable in a one-dimensional unate cascade.

Step 8: Steps 6 and 7 are repeated iteratively at each step obtaining a Φ_i function from Φ_{i-1} . If during any iteration, a function is obtained which is just a two variable function, we conclude f is unate cascade realizable; if not, f is not unate cascade realizable. The realizability test will be illustrated with the following examples:

Example 2.1: Let f be specified as a sum-of-products of prime implicants as

$$f = x_1 x_3 + x_2 x_3 + x_4$$

The product of the prime implicants is not equal to zero so f is unate.

The corresponding k -cube representation is $(2,2,3)$. The test steps

are as follows:

- 1) $(2,2,3)$ - Remove the 3-cube.
- 2) $(2,2)$ - No $n-1$ cubes, so add 1 to each k -cube.
- 3) $(3,3)$ - Possibly realizable because $(3,3)$ corresponds to $(n-1, n-1)$ and the number of steps is $n-1$.

$$4) \quad \Phi_1 = f = x_4 + x_1 x_3 + x_2 x_3$$

$$5) \quad \Phi_2 = x_3 (x_1 + x_2)$$

$$6) \quad \Phi_3 = x_1 + x_2$$

The unate cascade which realizes f is shown in Figure 2.5.

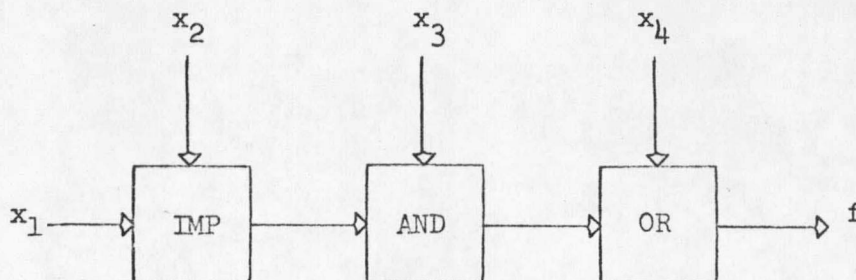


Figure 2.5 Unate Cascade Realizing $f = x_1 x_3 + x_2 x_3 + x_4$

Note that Example 2.1 is realizable only with variable orderings x_1, x_2, x_3, x_4 or x_2, x_1, x_3, x_4 . The function is not cascade realizable with any other ordering. This point is important because in later algorithms we are interested in realizability only with respect to a particular variable ordering.

Example 2.2: Let f be specified as

$$f = x_1 x_2 x_3 + x_2 x_3 x_4 + x_1 x_3 x_4$$

The function is unate and the k-cube representation is (1,1,1). The test steps are:

- 1) (1,1,1) - No n-1 cubes; add 1 to each k-cube.
- 2) (2,2,2) - No n-1 cubes; add 1 to each k-cube.
- 3) (3,3,3) - $n-1 = 4-1 = 3$ steps have been completed.

Since (3,3,3) does not correspond to (n-2) or (n-1,n-1), and n-1 steps have been taken, the function is not cascade realizable with any variable ordering and no further testing is necessary.

Example 2.3: Let f be specified as

$$f = x_1 x_2 + x_2 x_3 + x_3 x_4$$

The function is unate and the k-cube representation is (2,2,2). The

test steps are:

- 1) (2,2,2) - No n-1 cube. Add 1 to each k-cube.
- 2) (3,3,3) - Delete an n-1 cube.
- 3) (3,3) - Possibly realizable.
- 4) $\Phi_1 = f = x_1x_2 + x_2x_3 + x_3x_4$

The function f cannot be expressed in either form A or B of Step 6.

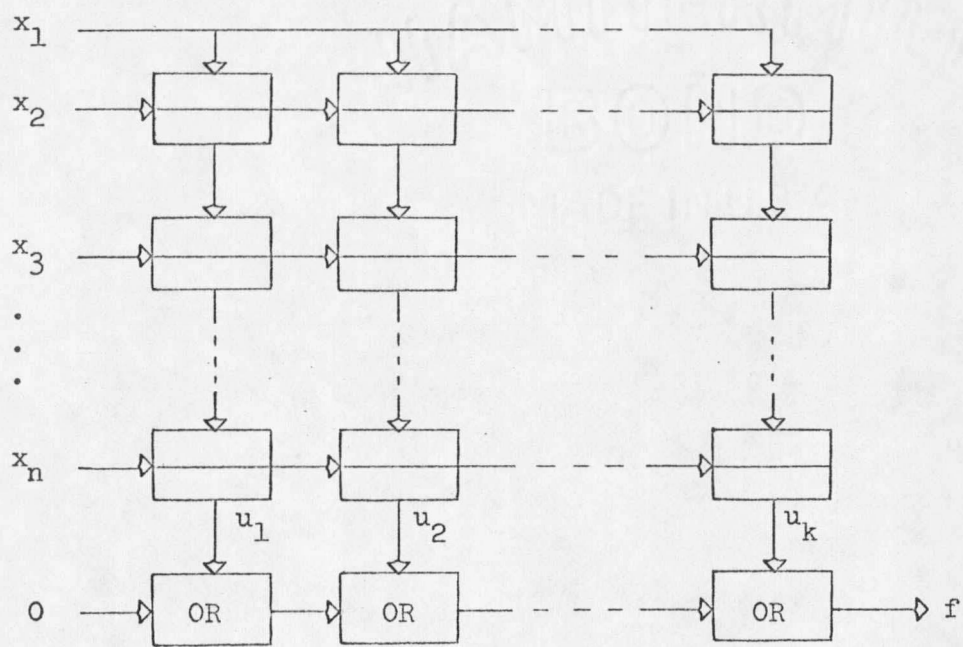
Therefore it is not realizable in a one-dimensional unate cascade.

This example shows that the realizability test condition is necessary, but not sufficient.

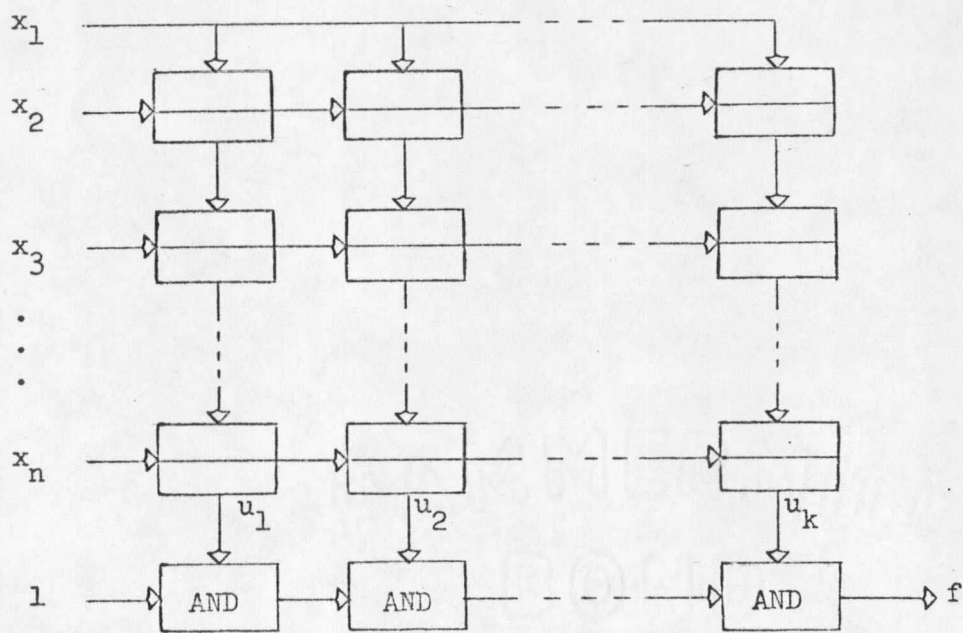
2.2.5 Two-Dimensional Arrays

Any arbitrary switching function $f(x_1, x_2, \dots, x_n)$ not realizable in a one-dimensional unate cascade can be realized by either one of the two-dimensional arrays shown in Figure 2.6 (a) and (b). The input variables are bussed horizontally. Each column of cells realizes a unate function of some or all of the input variables. The bottom row is a collector row which is a cascade of OR gates as in Figure 2.6 (a) or a cascade of AND gates as in Figure 2.6 (b).

A realization in the form of Figure 2.6 (a) is directly obtainable from the disjunctive canonical representation of f . Each funda-



(a)



(b)

Figure 2.6 Two-Dimensional Unate Cellular Array.

mental product term is a unate function and is unate cascade realizable. A cascade is needed for each fundamental product in f , the bottom row being employed to produce their disjunction. From dual arguments, a realization in the form of Figure 2.6 (b) is directly obtained from the conjunctive canonical form of f . In fact, any disjunctive or conjunctive normal form of f may be used to obtain such realizations from the minimal sum-of-products or product-of-sums forms.

The above procedure does not always guarantee the minimality of the number of unate cascades in the array. The minimization problem is essentially the following: a set of unate functions $u_1, u_2, \dots, u_k$ are needed such that

$$a) \text{ either } f = \sum_{i=1}^k u_i \text{ or } f = \prod_{i=1}^k u_i$$

b) each of the unate functions u_i is unate cascade realizable with the same ordering of the input variables.

c) k is a minimum

Since the two forms of f in a) are analogous, only the disjunctive form will be discussed.

2.3 Minimization Algorithm

An algorithm has been previously developed which obtains the minimalunate cascade realizations of an arbitrary switching function.⁽²⁸⁾ After a few definitions, this algorithm is given in Section 2.3.2. Most of the definitions and ideas developed in Sections 2.3.1 and 2.3.2 can be used to describe the improved minimization algorithm given in Section 2.5.

2.3.1 Definitions

The terms defined in this section are used in the algorithm given in Section 2.3.2 and in later sections.

Definition 2.1: If u is a unate function such that $u \subseteq f$ (u is included in f), then u is said to be a "unate component" of f . If $u \neq f$, then u is said to be properly included in f .

Definition 2.2: A unate component of f not properly included in any other unate component of f is said to be a "maximal unate component" of f .

Definition 2.3: Let $P = \{P_1, P_2, \dots, P_k\}$ be the set of prime implicants of f . A subset (proper or improper) of P , denoted as $P^j = (P_{j_1}, P_{j_2}, \dots, P_{j_s})$ is said to be "compatible" if and only if

$$\prod_{i=1}^s P_{j_i} \neq 0$$

Definition 2.4: A compatible set of prime implicants is said to be "maximal" if it is not a proper subset of any other compatible set of prime implicants.

Definition 2.5: Let $(p_1, p_2, \dots, p_k)$ be the set of product terms each of which is an implicant of the function f , but not necessarily a prime implicant of f . Let $P_{u_b} = (p_{i_1}, p_{i_2}, \dots, p_{i_m})$ be a maximal compatible set such that each p_{i_j} ($1 \leq j \leq m$) is a prime implicant of the function u_b defined as

$$u_b = \sum_{j=1}^m p_{i_j}$$

Then u_b will be called a "basic unate component" of f . Basic unate components include the maximal unate components and, in general, some non-maximal unate components.

Definition 2.6: A unate function obtained by dropping only one prime implicant from u_b will be called a "derived" unate component of f with respect to u_b . If there are m prime implicants of u_b , there will be m derived unate components with respect to u_b . The process can be iterated, that is, there can be derived unate components with respect to a previously derived unate component if necessary.

Definition 2.7: Let u_m be a basic or a derived unate component of f realizable by a unate cascade with a specified ordering of the in-

puts such that there does not exist another basic or derived unate component of f , say u_s , which properly includes u_m , that is $u_m \subset u_s$, and is realizable with the same ordering of the inputs. Then u_m is said to be a "prime unate" component of f with respect to the given ordering of the inputs.

Note that since non-prime cascades can always be replaced by prime cascades without destroying the covering of the function, only prime unate cascades need be considered in the minimization algorithm.

2.3.2 Algorithm Steps

The minimal unate cascade realization of an arbitrary switching function can be found as follows:

Step 1: Determine the basic unate components. Let $(p_1, p_2, \dots, p_k)$ be the set of implicants of the given function f . A compatibility graph G_c is then drawn as follows: G_c has a set of k vertices, one for each implicant term. Two vertices p_i and p_j are connected by an edge if and only if 1) $p_i p_j \neq 0$; that is, p_i and p_j are compatible; and 2) $p_i \not\subset p_j, p_j \not\subset p_i$; that is, p_i is not included in p_j and vice versa. The cliques* or maximal complete subgraphs of G_c correspond to the

*A clique of a linear non-directed graph is a maximal subgraph such that any pair of vertices within the subgraph is connected by an edge. A maximum clique is a clique having a maximum number of vertices. For further details see Chapter 4.

basic unate components of the given function.

Step 2: Set up a partially ordered graph for the basic unate components. Let $L_1 = (u_1, u_2, \dots, u_s)$ be the basic unate components of f . A partially ordered graph whose vertices correspond to the basic unate components is set up as in Figure 2.7. A line is drawn connecting two vertices if and only if one of the corresponding basic unate components is included in the other. Components which are not included in any other components correspond to the maximal unate components.

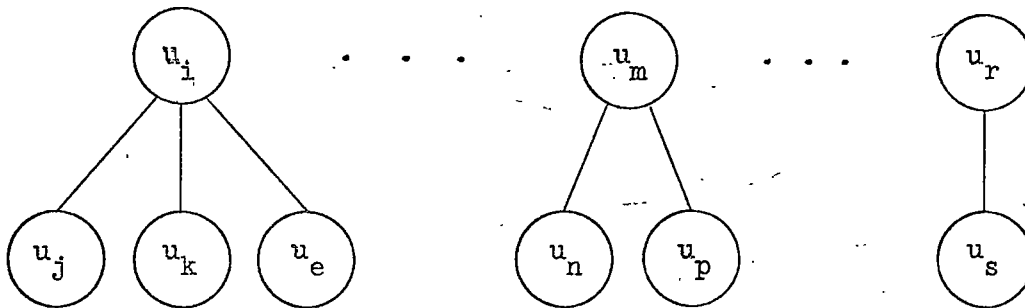


Figure 2.7 A Partially Ordered Graph

Step 3: Determine the prime unate components. For an arbitrarily specified ordering of the variables, the prime unate components of $L_1 = (u_1, u_2, \dots, u_s)$ which appear in the first level of the partially ordered graph of Figure 2.7 are tested for unate cascade realizability.

Let L_r and L_{nr} denote, respectively, the set of basic unate components on the first level of the partially ordered graph which are realizable* and non-realizable by unate cascades. Let L_c denote the set of unate components lying on the lower level of the partially ordered graph which are included in at least one element of L_r . Update list L_1 to L_1' as follows:

$$L_1' = L_1 - L_r - L_{nr} - L_c$$

If L_1' is empty, go to Step 4. If not, proceed as follows: Let u_b be an element of L_{nr} . Obtain the set of derived unate components with respect to u_b and consider only those which are not included by any element of L_r . Obtain such derived components for each of the elements of L_{nr} . Let L_d denote all such derived components. Update list L_1 to list L_2 as

$$L_2 = L_1' + L_d$$

A partially ordered graph (as in Figure 2.7) is now drawn with respect to L_2 and the above procedure is repeated. At some i th stage, L_i' must be empty since the number of derived components is finite.

Step 4: Determine a minimal cover. In Step 3 the list of prime unate components of f were obtained. A cover table for the fundamental products is now set up. A minimal set(s) of prime unate components are picked up which form a cover of the table. Then, for the assumed ordering of the variables, all minimal cascade realizations are obtained. These steps will be illustrated with an example in Section 2.5.2.

* Compared to the realizability test described in Section 2.2.3, the test used in the original algorithm (28) is much less sophisticated.

2.4

Disadvantages of the Previous Algorithm

The principal disadvantage of the algorithm presented in Section 2.3.2 lies in determining the basic unate components.

A computer program was written to implement the algorithm as it appears in Section 2.3.2 and it was found that approximately 80% of the execution time was required to just find the cliques of the graph of Step 1. The time cited was for a function of four variables which resulted in a compatibility graph with 20 vertices. The problem is that the time required to find the cliques increases almost exponentially with the number of vertices. For functions with more variables, a greater percentage of the execution time is needed to find the basic unate components.

Another disadvantage of the previous algorithm is that the process of setting up the partially ordered graph of Step 2 (Figure 2.7) may have to be repeated many times, depending on whether the basic unate components or derived components are unate cascade realizable or not.

In Section 2.5 an improved minimization algorithm is given which is an improvement over the algorithm of Section 2.3.2 for at least two reasons. First, the basic unate components are found by determining the cliques of a number of smaller graphs instead of solving one large graph. Second, using the results of Section 2.2.4, the improved algo-

ithm is set up so that only one partially ordered graph need be set up and the primeunate components can be picked from that graph.

2.5 Improved Minimization Algorithm

2.5.1 Detection of the Basic Unate Components

The following theorem will aid in finding the unate components of a function.

Theorem 2.5: Let $f(x_1, x_2, \dots, x_n)$ be expressed as a sum-of-products as $f = T_1 + T_2 + \dots + T_k$ where T_i , $1 \leq i \leq k$ are prime implicants of f . Then f is unate if and only if the product of the prime implicants $(\pi(T_1 T_2 \dots T_k))$ is not equal to zero.

Proof: If f is unate, then f is representable in a disjunctive normal formula in which no literal x_i appears both complemented and uncomplemented. Therefore, the logical product of the terms in the representation of f must not equal zero. On the other hand, if $f = T_1 + T_2 + \dots + T_k$ and $\pi(T_1 T_2 \dots T_k)$ is not equal to zero, no literal x_i appeared both primed and unprimed. Therefore, by definition, the function f must be unate.

This theorem gives a simple rule for testing for possible unate components. That is, two or more implicants can be combined to form a unate component if and only if they have some fundamental product terms in common (their logical product is not equal to zero).

To implement this idea an inclusion table can be set up in which each row of the table corresponds to one of the true fundamental products $(q_1, q_2, \dots, q_h)$ of the function, and each column corresponds to one of the implicants $(p_1, p_2, \dots, p_k)$ of the function. If the fundamental product q_i is included in the implicant p_j , an x is placed in the corresponding column of row q_i . By doing this, we have partitioned those implicants of the function which could possibly combine to form a basic unate component. Having a row in the table for each true fundamental product ensures that all possible combinations of implicants which could produce prime unate components will be considered.

After the inclusion table has been constructed, a compatibility graph can be set up for each row. The vertices of the graph G_i will correspond to the implicants appearing in row i . The rules for joining vertices are given in Step 1 of the algorithm in Section 2.3.2. The cliques of the graph G_i correspond to some of the basic unate components of the function. All basic unate components can be generated by computing the cliques for G_i , $1 \leq i \leq h$; (h is the number of true fundamental products in the function).

Note that each true fundamental product term is an implicant of the function. But since the fundamental product terms cannot combine with any other implicant of the function, the vertices in the compatibility graph corresponding to the true fundamental product terms will always be isolated, and therefore will always correspond to a basic unate component of the function. Knowing this fact, we can delete the fundamental product implicants of the function in the inclusion table and reduce the size of the table.

This process will be illustrated with respect to the function $f(w,x,y,z) = \sum (1,5,7,8,10,11,13,15)$. The implicants (not including fundamental products) which may be obtained by inspection from a Karnaugh map in this case, or by tabular methods⁽²²⁾ for more complicated examples, are: $p_1 = w'y'z$, $p_2 = xz$, $p_3 = xy'z$, $p_4 = w'xz$, $p_5 = xyz$, $p_6 = wxz$, $p_7 = wx'z'$, $p_8 = wyz$, and $p_9 = wx'y$.

The inclusion table for this example is given in Table 2.2.

Table 2.2 Inclusion Table for $f(w,x,y,z) = \sum (1,5,7,8,10,11,13,15)$

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
$q_1 = 1 = w'x'y'z$	x								
$q_2 = 5 = w'xy'z$	x	x	x	x					
$q_3 = 7 = w'xyz$		x		x	x				
$q_4 = 8 = wx'y'z'$							x		
$q_5 = 10 = wx'yz'$							x		x
$q_6 = 11 = wx'yz$								x	x
$q_7 = 13 = wxy'z$		x	x			x			
$q_8 = 15 = wxyz$		x			x	x		x	

At this point the inclusion table may possibly be reduced in size before continuing.

A row q_i of the inclusion table is said to dominate another row q_j of the same table (written $q_i \supset q_j$) if row q_i has x's in all the columns in which q_j has x's and if, in addition, row q_i has at least one x in a column in which row q_j does not have an x. The dominated row, q_j , can be removed because its implicants will be included in those of row q_i . If two rows q_i and q_j are identical, either may be removed because they both yield the same information.

If these two rules are applied to Table 2.2, rows q_1 and q_4 can be removed. The reduced table is shown in Table 2.3.

Table 2.3 Reduced Inclusion Table

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
$q_2 = 5$	x	x	x	x					
$q_3 = 7$		x		x	x				
$q_5 = 10$							x		x
$q_6 = 11$								x	x
$q_7 = 13$		x	x			x			
$q_8 = 15$		x			x	x		x	

The basic unate components can now be determined. From earlier considerations the first eight basic unate components (corresponding to the fundamental products) are $u_i = q_i$ for $1 \leq i \leq 8$.

Figure 2.8 (a) and (b) show the compatibility graphs corresponding to the six rows of Table 2.3 (see rules for joining the vertices of the compatibility graph in Step 1 of Section 2.3.2).

The basic unate components are tabulated in Table 2.4. Components u_9 through u_{16} correspond to the cliques of Figure 2.8 (a) and (b).

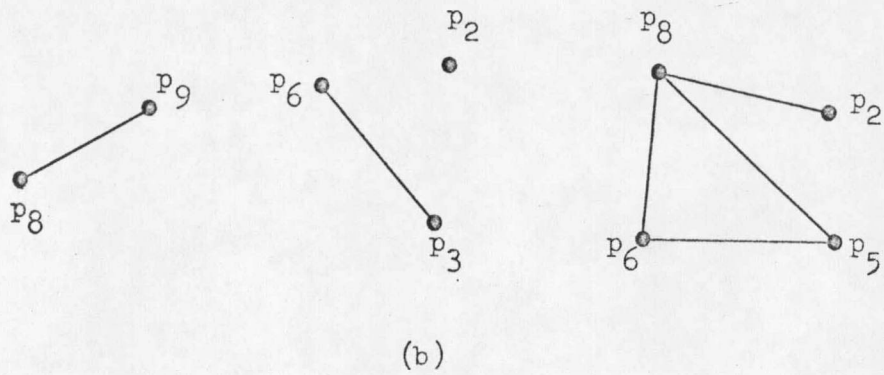
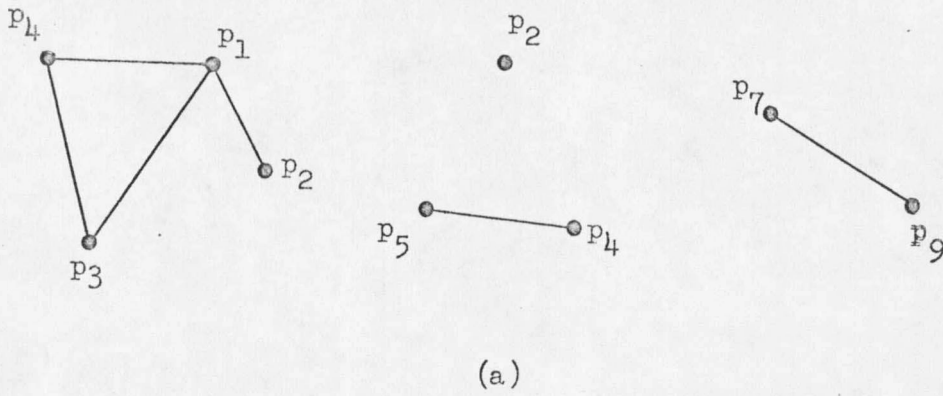


Figure 2.8 Compatibility Graphs for Table 2.3.

Table 2.4 Basic Unate Components

$u_1 = q_1$	$u_9 = p_1 + p_2$
$u_2 = q_2$	$u_{10} = p_1 + p_3 + p_4$
$u_3 = q_3$	$u_{11} = p_4 + p_5$
$u_4 = q_4$	$u_{12} = p_7 + p_9$
$u_5 = q_5$	$u_{13} = p_8 + p_9$
$u_6 = q_6$	$u_{14} = p_3 + p_6$
$u_7 = q_7$	$u_{15} = p_2 + p_8$
$u_8 = q_8$	$u_{16} = p_5 + p_6 + p_8$

2.5.2 Improved Algorithm Steps

The steps for the improved minimization algorithm can now be stated. As an example, the minimal cascade realization for $f(w,x,y,z) = \sum (1,5,7,8,10,11,13,15)$ with variable ordering $yxzw$ will be found.

Step 1: Determine the basic unate components. This step has been explained in detail in Section 2.5.1.

Step 2: Test and derive basic unate components. All the prime unate components which will appear in the minimal cascade realization will come from the basic unate components (some may be derived components).

Each basic unate component is tested for unate cascade realizability with the desired variable ordering using the realizability test of Section 2.2.4. If a basic unate component is not unate cascade realizable, it is replaced by its derived components and they are tested as in Section 2.2.4. If the derived components are not unate cascade realizable, they are replaced by their derived components and so on. (This process must terminate since the number of derived components is finite.) The deriving process is carried on until all the derived components are unate cascade realizable.

Step 3: When all the basic unate components have been so tested, a partially ordered graph (see Step 2 of Section 2.3.2) is set up with all the unate cascade realizable components generated in Step 2. The prime unate components correspond to those components which are not included in any other component in the partially ordered graph.

Step 4: A cover table is set up with the prime unate components determined in Step 3. A minimal set(s) of prime unate components are picked which form a cover of the table. Then, for the assumed ordering of the variables, all minimal cascade realizations are obtained.

For our example, the basic unate components appear in Table 2.4. Components u_1 through u_8 are obviously unate cascade realizable since they consist of only one term. Testing components u_9 through u_{16} with

the variable ordering $yxzw$ in Section 2.2.4, we get the following unate cascade realizable functions:

$$\begin{array}{ll}
 u_{17} = p_1 & \left. \begin{array}{l} \\ \\ \\ \end{array} \right\} \text{derived from } u_9 \\
 u_{18} = p_2 & \\
 \\
 u_{19} = p_3 & \left. \begin{array}{l} \\ \\ \end{array} \right\} \text{derived from } u_{10} \\
 u_{20} = p_1 + p_4 & \\
 \\
 u_{21} = p_5 & \left. \begin{array}{l} \\ \end{array} \right\} \text{derived from } u_{11} \\
 \\
 u_{22} = p_7 & \left. \begin{array}{l} \\ \\ \end{array} \right\} \text{derived from } u_{12} \\
 u_{23} = p_9 & \\
 \\
 u_{24} = p_8 & \left. \begin{array}{l} \\ \end{array} \right\} \text{derived from } u_{13} \\
 \\
 u_{25} = p_6 & \left. \begin{array}{l} \\ \end{array} \right\} \text{derived from } u_{14} \\
 \\
 u_{26} = p_6 + p_8 & \left. \begin{array}{l} \\ \end{array} \right\} \text{derived from } u_{16}
 \end{array}$$

The unate cascade realizable components are u_1 through u_8 and u_{17} through u_{26} . The partially ordered graph for these components appears in Figure 2.9.

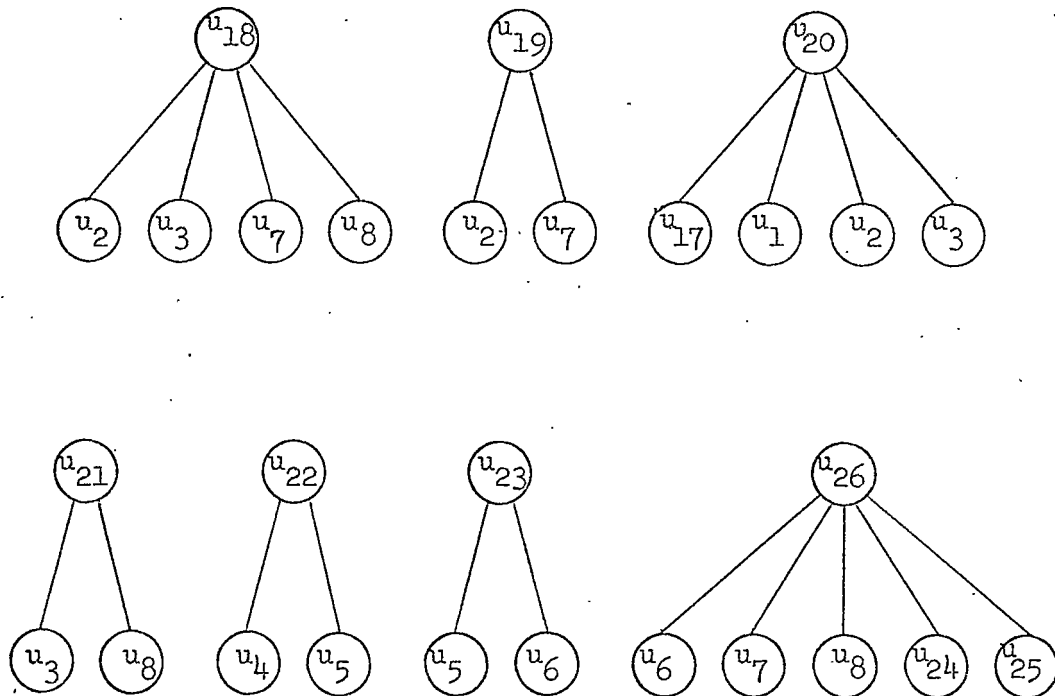


Figure 2.9 Partially Ordered Graph for Unate Cascade Realizable Components

From Figure 2.9, the prime unate components are u_{18} , u_{19} , u_{20} , u_{21} , u_{22} , u_{23} , and u_{26} . A cover table (Table 2.5) can now be set up for determining the minimal solution(s).

Components u_{20} and u_{22} are essential for the solution since they have columns with a single x. Deleting u_{20} and u_{22} , the table can be reduced and appears in Table 2.6.

Table 2.5 Cover Table for $f = \sum (1,5,7,8,10,11,13,15)$

	1	5	7	8	10	11	13	15
u_{18}		x	x				x	x
u_{19}		x					x	
$*u_{20}$	x	x	x					
u_{21}			x					x
$*u_{22}$				x	x			
u_{23}					x	x		
u_{26}						x	x	x

Table 2.6 Reduced Cover Table

	11	13	15
u_{18}		x	x
u_{19}		x	
u_{21}			x
u_{23}	x		
u_{26}	x	x	x

In the reduced table, u_{26} is the obvious choice for inclusion in the minimal solution because it covers the three remaining fundamental products 11, 13, and 15. For more complicated cover tables, general algorithms are developed elsewhere. (2) (22) (23)

The minimal solution for $f(w,x,y,z) = \sum (1,5,7,8,10,11,13,15)$ for variable ordering y,x,z,w is

$$f = u_{20} + u_{22} + u_{26}$$

$$= w'z (x+y') + wz'x' + wz(x+y).$$

The circuit which realizes f is shown in Figure 2.10.

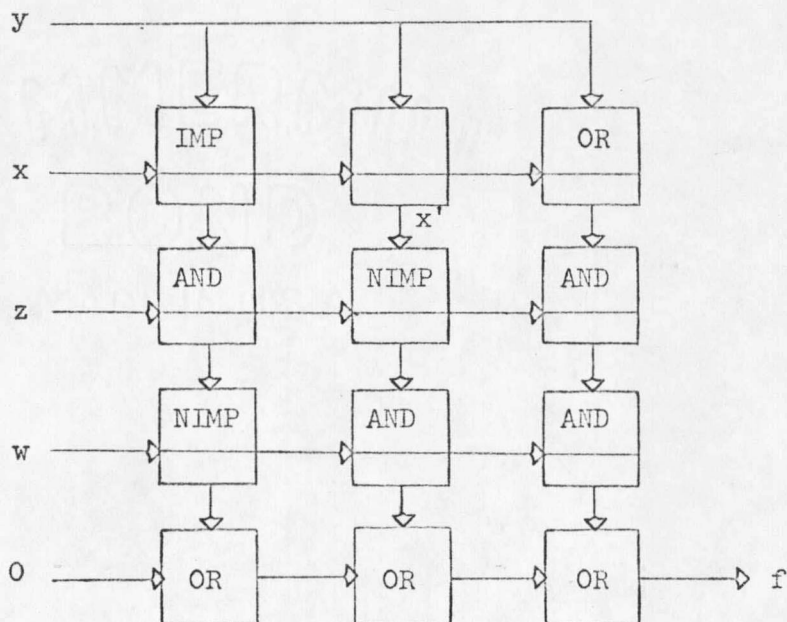


Figure 2.10 Minimal Cascade Realization

2.6 Algorithm Extensions

2.6.1 Optional States

For functions with optional states, there are two minor modifications which must be made in the algorithm of Section 2.5. First, any optional states which are present are treated as true (1) terms in forming the implicants, except that no implicants are formed which contain only optional states. Second, optional states are not included in the cover table to determine the minimal sum(s).

2.6.2 Multiple Output Functions

2.6.2.1 Introduction

Combinational networks are sometimes required which have more than one output. Each output is a function of the same set of input variables.

A set of functions $f_1, f_2, \dots, f_k$ can be synthesized using a two-dimensional array as shown in Figure 2.11. The design of the collector array is identical to that suggested by Minnick.⁽²⁴⁾ For the disjunctive realization, the minimization problem amounts to finding a set ofunate functions $u_1, u_2, \dots, u_m$ such that:

- a) Each of the unate functions is cascade realizable with the same ordering of the input variables.

b) Each output function $f_1, f_2, \dots, f_k$ can be expressed as a sum of some or all of the unate cascade realizable functions $u_1, u_2, \dots, u_m$.

c) m is minimum.

Perhaps the most obvious approach to finding the minimal solution to the multiple output case would be to minimize the functions separately and realize each function independently of the others. While this approach is straight-forward, it would not always yield a minimal solution because there could be cascade realizable unate components common to a number of output functions which could be used in the minimal realization. These "shared" unate components must also be considered for possible inclusion in the minimal solution. For three functions, say f_1 , f_2 , and f_3 , we must find the cascade realizable unate components which are common to f_1 and f_2 , f_1 and f_3 , f_2 and f_3 , and f_1 , f_2 , and f_3 . This implies that we know which implicants are common among the functions.

2.6.2.2

Determination of Common Unate Components

There exist tabular methods⁽²²⁾ of determining the implicants of a set of functions $f_1, f_2, \dots, f_k$. Consider the set of functions $f_1(w, x, y, z) = \sum (0, 1, 5, 6, 8, 12, 13, 14)$, $f_2(w, x, y, z) = \sum (5, 6, 7, 12, 13, 15)$, and $f_3(w, x, y, z) = \sum (0, 1, 6, 7, 8, 14, 15)$. The set of im-

plicants (except fundamental product implicants) for these functions are tabulated in Table 2.7. The implicant is given together with its corresponding decimal equivalent and a tag field which indicates in which functions the implicants are included. A (-) implies the implicant is included in that function; a 0 implies that it is not.

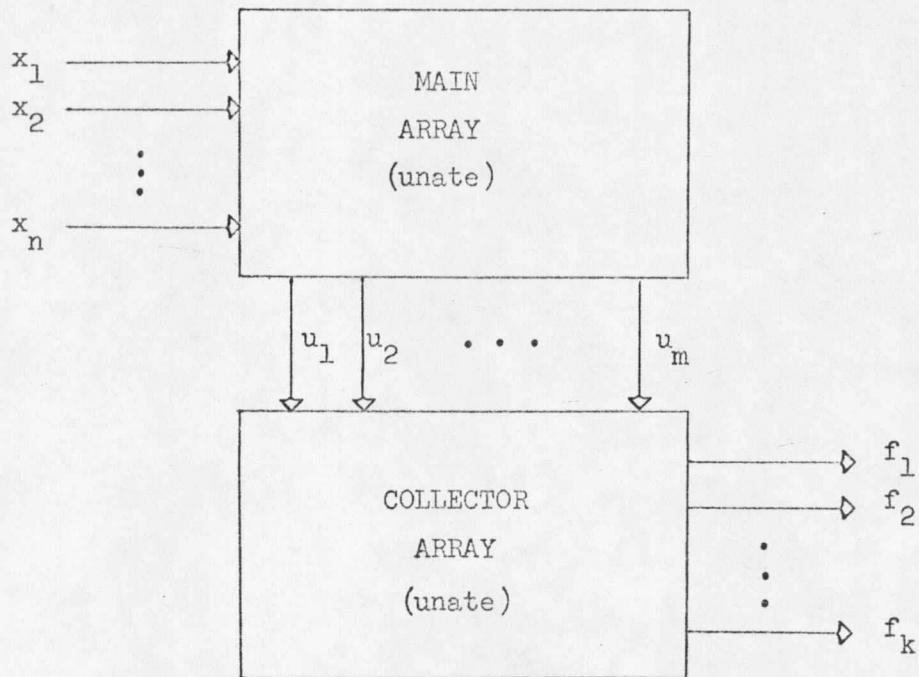


Figure 2.11 Multiple-Output Two-Dimensional Unate Cellular Array

Table 2.7 Implicants for f_1 , f_2 , and f_3

	<u>Decimal Equivalent</u>	<u>f_1</u>	<u>f_2</u>	<u>f_3</u>
$p_1 = w'x'y'$	(0,1)	-	0	-
$p_2 = x'y'z'$	(0,8)	-	0	-
$p_3 = w'y'z$	(1,5)	-	0	0
$p_4 = wy'z'$	(8,12)	-	0	0
$p_5 = w'xz$	(5,7)	0	-	0
$p_6 = xy'z$	(5,13)	-	-	0
$p_7 = w'xy$	(6,7)	0	-	-
$p_8 = xyz'$	(6,14)	-	0	-
$p_9 = wxy'$	(12,13)	-	-	0
$p_{10} = wxz'$	(12,14)	-	0	0
$p_{11} = xyz$	(7,15)	0	-	-
$p_{12} = wxz$	(13,15)	0	-	0
$p_{13} = wxy$	(14,15)	0	0	-
$p_{14} = xz$	(5,7,13,15)	0	-	0
$p_{15} = xy$	(6,7,14,15)	0	0	-

To find the minimal set of unate cascade realizable functions $u_1, u_2, \dots, u_m$ of Figure 2.7 needed to realize the set of functions $f_1, f_2, \dots, f_k$, all possible unate cascade realizable components which could be used in the minimal solution must be determined.

First, the prime unate components of the functions $f_1, f_2, \dots, f_k$ are found. Next, the prime unate components common to f_1 and f_2 , f_1 and $f_3, \dots, f_{k-1}$ and f_k are found, and so on. Finally, the prime unate components common to f_1 and f_2 , and, ..., and f_k are found.

A multiple-output cover table⁽²⁾⁽²²⁾⁽²³⁾ is then set up (analogous to the multiple-output prime implicant table) and the minimal covers are determined.

For example, to determine the prime unate components common to f_1 and f_3 , we would consider all the implicants of Table 2.7 common to f_1 and f_3 (these have - 0 - or - - - in the tag field). They are p_1 , p_3 , and p_8 .

Using the implicants p_1 , p_2 , and p_8 , and the true fundamental product terms common to f_1 and f_3 (0,1,6,8,14), an inclusion table can be set up as in Section 2.5 and the basic unate components determined. The inclusion table is shown in Figure 2.12.

	p_1	p_2	p_8
0	x	x	
1	x		
6			x
8		x	
14			x

Figure 2.12 Inclusion Table for f_1 and f_3

Rows 1, 8, and 14 can be eliminated, leaving only rows 0 and 6 which yield the basic unate components $p_1 + p_2$ and p_8 .

If a variable ordering, say wzyx, is specified, the prime unate components common to f_1 and f_3 may be found. They are just the basic unate components in this case since both are realizable with the specified ordering.

In Table 2.8, the prime unate components, their decimal equivalents, and the functions to which they belong are tabulated.

Using the prime unate components of Table 2.8, a multiple output cover table for f_1 , f_2 , and f_3 can be set up. (Table 2.9).

Table 2.8 Prime Unate Components for f_1 , f_2 , and f_3

	<u>Decimal Equivalent</u>	<u>Function</u>
$u_1 = p_1 + p_2$	(0,1,8)	f_1
$u_2 = p_6 + p_9$	(5,12,13)	f_1
$u_3 = p_3$	(1,5)	f_1
$u_4 = p_4$	(8,12)	f_1
$u_5 = p_{10}$	(12,14)	f_1
$u_6 = p_8$	(6,14)	f_1
$u_7 = p_{14}$	(5,7,13,15)	f_2
$u_8 = p_7 + p_{11}$	(6,7,15)	f_2
$u_9 = p_6 + p_9$	(5,12,13)	f_2
$u_{10} = p_1 + p_2$	(0,1,8)	f_3
$u_{11} = p_{15}$	(6,7,14,15)	f_3
$u_{12} = p_6 + p_9$	(5,12,13)	f_1 and f_2
$u_{13} = \text{fund. product}$	(6)	f_1 and f_2
$u_{14} = p_1 + p_2$	(0,1,8)	f_1 and f_3
$u_{15} = p_8$	(6,14)	f_1 and f_3
$u_{16} = p_7 + p_{11}$	(6,7,15)	f_2 and f_3
$u_{17} = \text{fund. product}$	(6)	f_1, f_2 and f_3

Table 2.9 Multiple Output Cover Table

	f_1								f_2							f_3						
	0	1	5	6	8	12	13	14	5	6	7	12	13	15	0	1	6	7	8	14	15	
u_1	x	x			x																	
u_2			x			x	x															
u_3		x	x																			
u_4					x	x																
u_5						x		x														
u_6				x				x														
u_7									x		x		x	x								
u_8										x	x			x								
u_9									x			x	x									
u_{10}															x	x			x			
u_{11}																	x	x		x	x	
u_{12}			x			x	x		x			x	x									
u_{13}				x						x												
u_{14}	x	x			x										x	x			x			
u_{15}			x					x									x			x		
u_{16}										x	x			x			x	x			x	
u_{17}				x						x							x					

If the cost of each row in Table 2.9 is assumed equal, the prime unate components u_{12} , u_{14} , u_{15} , and u_{16} form a minimal set which can be used to realize f_1 , f_2 , and f_3 . The covers are

$$f_1 = u_{12} + u_{14} + u_{15} = xy'(z+w) + x'y'(z+w') + xyz'$$

$$f_2 = u_{12} + u_{16} = xy'(z+w) + xy(z+w')$$

$$f_3 = u_{14} + u_{15} + u_{16} = x'y'(z+w') + xyz' + xy(z+w').$$

The circuit which realizes f_1 , f_2 , and f_3 is shown in Figure 2.13.

2.7 Variations of Unate Cascade Results

The two types of unate cascade realizations obtained thus far have used the OR operator or, by duality, the AND operator in the collector row of Figure 2.6 (a) and (b). The most general type of array would allow arbitrary cell functions in the collector row as well as in the rest of the array. However, the minimization problems associated with arrays of this nature are very complex and no attempt is made to solve them here. Instead, attention is concentrated on realizations in which all cells of the collector row are identical.

If a collector row of $\text{NOR}(x'y')$, $\text{NAND}(x+y')$, $\text{IMP}(x+y', x'+y)$, or $\text{NIMP}(xy', x'y)$ cells is allowed, only the realizations with $\text{IMP}(x+y')$ or $\text{NIMP}(xy')$ operators in the collector row are directly obtainable from

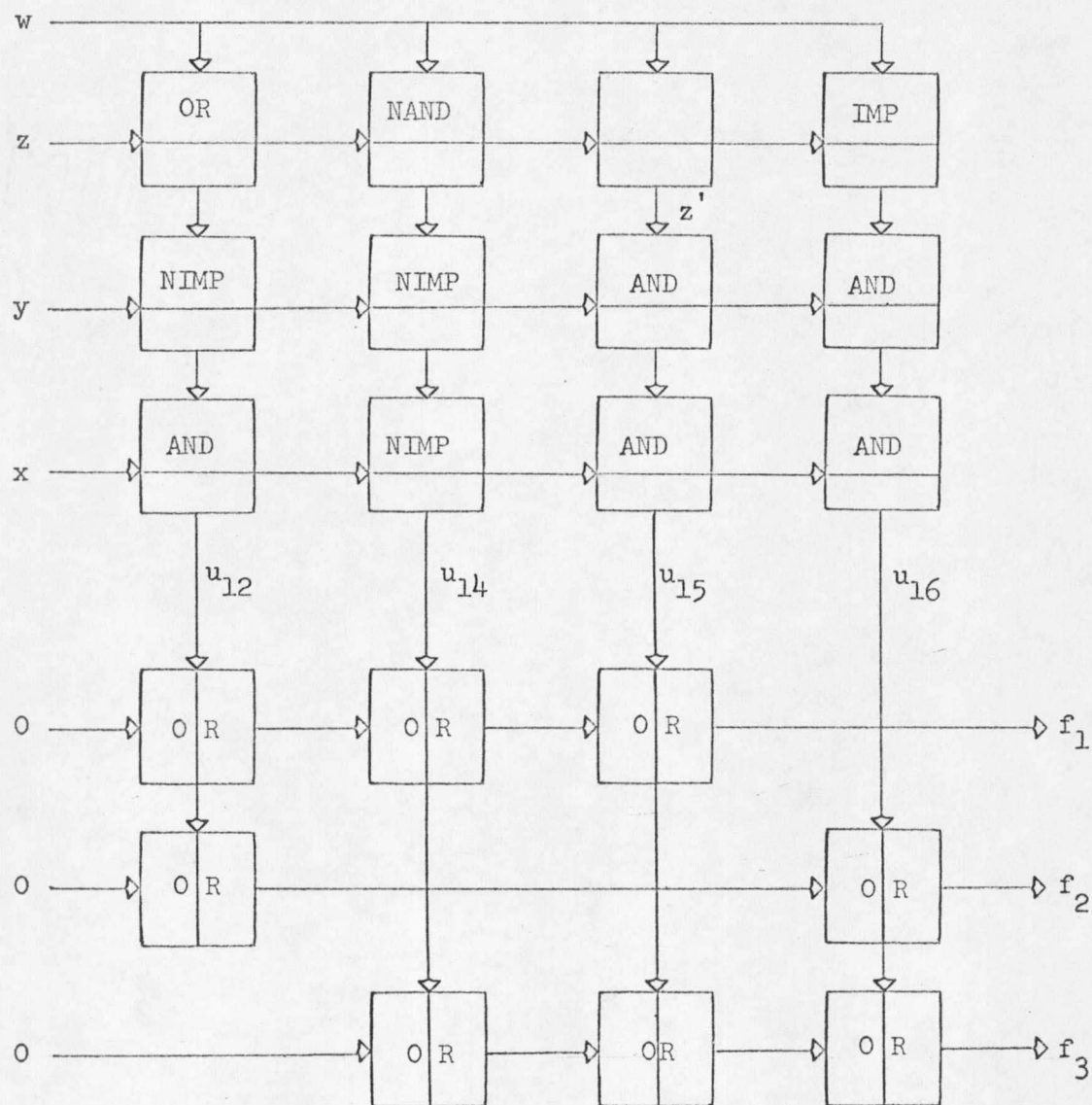


Figure 2.13 Multiple-Output Realization

the results of this chapter. The problem of obtaining realizations using $\text{NOR}(x' y')$, $\text{NAND}(x' + y')$, $\text{IMP}(x' + y)$, or $\text{NIMP}(x' y)$ collector row operators involves two-level expressions of unate functions and further research is required to solve these problems.

There are two other nontrivial functions that could be used for the collector row operator, Exclusive Or and Equivalence. The problem of finding a realization using unate cascade functions and an Exclusive Or collector row has been partially solved in the case where the functions form a subset of the positive unate functions or q-functions as they are more commonly known in the literature.⁽²⁷⁾ The q-function consists of a cascade of majority cells whose output ($z = xy + bx + by$) is determined by the b parameter which can be set so that the cell produces either the OR or the AND function.

Simplification algorithms for q-functions connected by a row of Exclusive Or cells are available in the literature⁽²⁷⁾ although they do not always lead to minimal solutions.

The q-functions only cover a small class of realizations involving the Exclusive Or operator. Other variations in which the cascades produce arbitrary unate functions and the collector row is composed of Exclusive Or cells can be considered. The solutions to these problems are fundamentally related to the solutions of some minimization problems which can be formulated using the Exclusive Or and Equivalence operators. This will be the subject matter of the next chapter.

Chapter 3

MINIMIZATION OF CELLULAR ARRAYS BASED ON REED-MULLER CANONIC FORMS

3.1 Introduction

In Chapter 2, unate cascades were used to realize arbitrary switching functions. For a fixed variable ordering, algorithms were developed to find the minimal size array needed to synthesize any function. The realizations of the previous chapter were based on a form of Shannon's ⁽²²⁾ disjunctive or conjunctive canonical representation of the function.

In search of more efficient cellular arrays, this chapter develops minimization algorithms based on the Reed-Muller canonic expansions of a function. After a few preliminary definitions, the minimization problems are more fully discussed.

3.2 Elementary Definitions and Canonical Forms.

This section will review briefly the canonic representations of a switching function f using EXCLUSIVE OR (symbolized as $\oplus$) and LOGICAL EQUIVALENCE (symbolized as $\odot$) operations as primary operators and LOGICAL AND (symbolized as $\cdot$, to be omitted for convenience) and LOGICAL OR (symbolized as $+$) as secondary operators, with or without complementation (symbolized $'$) of input variables $x_1, x_2, \dots, x_n$.

Some of the important and useful theorems with respect to these operations are:

$$x \oplus y = xy' + x'y$$

$$x \odot y = xy + x'y'$$

$$x \oplus x = 0$$

$$x \odot x = 1$$

$$x \oplus x' = 1$$

$$x \odot x' = 0$$

$$x \oplus 0 = x$$

$$x \odot 0 = x'$$

$$x \oplus 1 = x'$$

$$x \odot 1 = x$$

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

$$x \odot (y \odot z) = (x \odot y) \odot z$$

$$x(y \oplus z) = xy \oplus xz$$

$$x + (y \odot z) = (x + y) \odot (x + z)$$

If $f = x_1 \oplus x_2 \oplus \dots \oplus x_n$, then $f = 1$ if an odd number of the variables $x_1, \dots, x_n$ have value 1. If $f = x_1 \odot x_2 \odot x_3 \odot \dots \odot x_n$, then $f = 1$ if an even number of the variables $x_1, x_2, \dots, x_n$ have value 0. Finally, if a function f can be expressed as

$$f = P_1 + P_2 + \dots + P_k \quad (3.1)$$

$$= Q_1 Q_2 \dots Q_m \quad (3.2)$$

where P_i ($1 \leq i \leq k$) and Q_j ($1 \leq j \leq m$) are arbitrary functions such that $P_i P_j = 0$ and $Q_i Q_j = 0$, for all i and j , $i \neq j$, then

$$f = P_1 \oplus P_2 \oplus \dots \oplus P_k \quad (3.3)$$

$$= Q_1 \odot Q_2 \odot \dots \odot Q_m \quad (3.4)$$

A variable x_i ($1 \leq i \leq n$) in the following expansions can be either x_i or x_i' ; the symbols $a_i, b_i, c_i, \alpha_i, \beta_i$, and δ_i are understood to be the binary constants 0 or 1. The expansions are written for $n = 2, 3$ but the

general forms for arbitrary n should be obvious.

The following are the Exclusive Or canonical forms:

$$\begin{aligned}
 (a) \quad f(x_1, x_2, x_3) &= a_0 \oplus a_1 \dot{x}_1 \oplus a_2 \dot{x}_2 \oplus a_3 \dot{x}_3 \oplus a_4 \dot{x}_1 \dot{x}_2 \oplus \\
 &\quad a_5 \dot{x}_1 \dot{x}_3 \oplus a_6 \dot{x}_2 \dot{x}_3 \oplus a_7 \dot{x}_1 \dot{x}_2 \dot{x}_3 \quad (3.5)
 \end{aligned}$$

This expression is called the complement-free ring sum expansion or the Reed-Muller expansion when all the $\dot{x}_i$ are x_i .

$$\begin{aligned}
 (b) \quad f(x_1, x_2, x_3) &= (b_0 + 0) \oplus (b_1 + \dot{x}_1) \oplus (b_2 + \dot{x}_2) \oplus (b_3 + \dot{x}_3) \\
 &\quad \oplus (b_4 + \dot{x}_1 + \dot{x}_2) \oplus (b_5 + \dot{x}_1 + \dot{x}_3) \\
 &\quad \oplus (b_6 + \dot{x}_2 + \dot{x}_3) \oplus (b_7 + \dot{x}_1 + \dot{x}_2 + \dot{x}_3) \quad (3.6)
 \end{aligned}$$

$$\begin{aligned}
 (c) \quad f(x_1, x_2) &= c_0 \oplus c_1 x_1 \oplus c_2 x_1' \oplus c_3 x_2 \oplus c_4 x_1 x_2 \\
 &\quad \oplus c_5 x_1' x_2' \oplus c_6 x_2' \oplus c_7 x_1 x_2' \oplus c_8 x_1' x_2' \quad (3.7)
 \end{aligned}$$

The Logical Equivalence canonical forms are as follows:

$$\begin{aligned}
 (a) \quad f(x_1, x_2, x_3) &= (\alpha_0 + 0) \odot (\alpha_1 + \dot{x}_1) \odot (\alpha_2 + \dot{x}_2) \\
 &\quad \odot (\alpha_3 + \dot{x}_3) \odot (\alpha_4 + \dot{x}_1 + \dot{x}_2) \\
 &\quad \odot (\alpha_5 + \dot{x}_1 + \dot{x}_3) \odot (\alpha_6 + \dot{x}_2 + \dot{x}_3) \\
 &\quad \odot (\alpha_7 + \dot{x}_1 + \dot{x}_2 + \dot{x}_3) \quad (3.8)
 \end{aligned}$$

$$(b) \quad f(x_1, x_2, x_3) = \beta_0 \odot \beta_1 \dot{x}_1 \odot \beta_2 \dot{x}_2 \odot \beta_3 \dot{x}_3 \odot \beta_4 \dot{x}_1 \dot{x}_2 \\ \odot \beta_5 \dot{x}_1 \dot{x}_3 \odot \beta_6 \dot{x}_2 \dot{x}_3 \odot \beta_7 \dot{x}_1 \dot{x}_2 \dot{x}_3 \quad (3.9)$$

$$(c) \quad f(x_1, x_2) = (\delta_0 + 0) \odot (\delta_1 + x_1) \odot (\delta_2 + \dot{x}_1) \odot (\delta_3 + x_2) \\ \odot (\delta_4 + x_1 + x_2) \odot (\delta_5 + \dot{x}_1 + x_2) \odot (\delta_6 + \dot{x}_2) \\ \odot (\delta_7 + x_1 + x_2) \odot (\delta_8 + \dot{x}_1 + \dot{x}_2) \quad (3.10)$$

Expansions (3.8), (3.9), and (3.10) are the dual forms of the expansions (3.5), (3.6), and (3.7), respectively. Expansions (3.5), (3.6), (3.8), and (3.9) are unique for any given function with respect to given set of input variables $\dot{x}_1, \dot{x}_2, \dots, \dot{x}_n$, whereas expansions (3.7) and (3.10) are not unique. Given a function in the well-known Shannon form

$$f(x_1, \dots, x_n) = \sum_{i=0}^{2^n-1} f_i p_i = \prod_{i=0}^{2^n-1} (s_i + f_{2^{n-1}-i}) \quad (3.11)$$

where p_i and s_i are the fundamental product (minterm) or fundamental sum (maxterm) respectively associated with the i th configurations of $(x_1, x_2, \dots, x_n)$, and f_i and $f_{2^{n-1}-i}$ are binary constants, the procedures to obtain the unique canonic forms described above are as follows:

To obtain expansion (3.5), using equations (3.3) and (3.11), we write

$$f = \bigoplus_{i=0}^{2^n-1} f_i p_i \quad (3.12)$$

where $\bigoplus \sum$ denotes sum mod 2. For all i such that $f_i = 1$, we can express p_i without loss of generality as

$$\begin{aligned} p_i &= \dot{x}_1 \dot{x}_2 \dots \dot{x}_s \dot{x}_{s+1} \dot{x}_{s+2} \dots \dot{x}_n \\ &= \dot{x}_1 \dot{x}_2 \dots \dot{x}_s (1 \oplus \dot{x}_{s+1}) (1 \oplus \dot{x}_{s+2}) \dots (1 \oplus \dot{x}_n) \end{aligned} \quad (3.13)$$

Factoring out p_i and then substituting in equation (3.12) and simplifying using $x \oplus x = 0$ and $0 \oplus x = x$, we obtain the expansion (3.5). We could have obtained the same expansion starting from any other arbitrary expression of f in the form given by equation (3.3). To obtain the expansion (3.9) of f , we need to prove the following Theorem.

Theorem 3.1: Let there be T product terms present including the constant term a_0 ($a_0 = 0$ is trivial) in the representation of a function f as

$$f = a_0 \oplus a_1 \dot{x}_1 \oplus a_2 \dot{x}_2 \oplus \dots \oplus a_{2^n-1} \dot{x}_1 \dot{x}_2 \dots \dot{x}_n \quad (3.14)$$

then f can also be represented as

$$f = t \odot a_1 \dot{x}_1 \odot a_2 \dot{x}_2 \odot \dots \odot a_{2^n-1} \dot{x}_1 \dot{x}_2 \dots \dot{x}_n \quad (3.15)$$

where $t = 1$ when T is odd and $t = 0$ when T is even.

Conversely, let there be D product terms present including the constant term a_0 ($a_0 = 1$ is trivial) in the representation of a function as

$$f = a_0 \odot a_1 \dot{x}_1 \odot a_2 \dot{x}_2 \odot \dots \odot a_{2^n-1} \dot{x}_1 \dot{x}_2 \dots \dot{x}_n \quad (3.16)$$

then f can also be represented as

$$f = d \oplus a_1 \dot{x}_1 \oplus a_2 \dot{x}_2 \oplus \dots \oplus a_{2^n-1} \dot{x}_1 \dot{x}_2 \dots \dot{x}_n \quad (3.17)$$

where $d = 0$ when D is odd and $d = 1$ when D is even.

Proof: First let T be odd and assume that $f = 1$ for some assignment of input variables. If $a_0 = 0$, an odd number of terms in (3.14), say W , must be 1 and therefore an even number of terms $(T-W)$ in (3.15) become 0 making (3.15) a valid representation of f . If $f = 0$ for this assignment, an even number of terms in (3.14), say U , must be 1 and hence an odd number $(T-U)$ of terms in (3.15) must be 0 making (3.15) still a valid representation. Now assume $a_0 = 1$ and $f = 1$. Then an even number, say V , of $(T-1)$ non-trivial terms must be 1 and an even number of terms $(T-1-V)$ in (3.15) become 0. If for the same variable assignment $f = 0$, an odd number, say Q , of $(T-1)$ non-trivial terms must be 1 and an odd number $(T-1-Q)$ of terms in (3.15) become 0. Therefore, (3.15) is a valid representation when T is odd. From similar arguments, (3.15) can be

derived when T is even. The proof of the converse follows almost identical reasoning and will be omitted.

To obtain expansion (3.8) using equations (3.4) and (3.11) we write

$$f = \odot \sum_{i=0}^{2^n-1} (s_i + f_{2^{n-1-i}})$$

where $\odot \sum$ denotes complement sum mod 2. For all i such that $f_{2^{n-1-i}} = 0$, we can express s_i without loss of generality as

$$\begin{aligned} s_i &= \dot{x}_1 + \dot{x}_2 + \dots + \dot{x}_s + \dot{x}_{s+1} + \dot{x}_{s+2} + \dots + \dot{x}_n \\ &= \dot{x}_1 + \dot{x}_2 + \dots + \dot{x}_s + (0 \odot \dot{x}_{s+1}) + \dots + (0 \odot \dot{x}_n) \end{aligned}$$

Applying the distributive law $a + (b \odot c) = (a + b) \odot (a + c)$, substituting in f and simplifying using $x \odot x = 1$ and $1 \odot x = x$, we obtain expansion (3.8). We could have obtained the same expansion starting from any other arbitrary expression of f , as given by (3.4). To obtain expansion (3.6) of f , we use Theorem 3.2, stated below, whose proof follows the same line of arguments used to prove Theorem 3.1.

Theorem 3.2: Let there be T sum terms present including the constant term α_0 ($\alpha_0 = 1$ is trivial) in the representation of f as

$$f = (\alpha_0 + 0) \odot (\alpha_1 + \dot{x}_1) \odot (\alpha_2 + \dot{x}_2) \odot \dots \odot (\alpha_{2^n-1} + \dot{x}_1 + \dot{x}_2 + \dots + \dot{x}_n)$$

then f can also be represented as

$$f = t \oplus (\alpha_1 + \dot{x}_1) \oplus (\alpha_2 + \dot{x}_2) \oplus \dots \oplus (\alpha_{2^n-1} + \dot{x}_1 + \dot{x}_2 + \dots + \dot{x}_n)$$

where $t = 0$ when T is odd and $t = 1$ when T is even.

Conversely, let there be D sum terms present including the constant term α_0 ($\alpha_0 = 0$ is trivial) in the representation of a function as

$$f = (\alpha_0 + 0) \oplus (\alpha_1 + \dot{x}_1) \oplus (\alpha_2 + \dot{x}_2) \oplus \dots \oplus (\alpha_{2^n-1} + \dot{x}_1 + \dot{x}_2 + \dots + \dot{x}_n)$$

then f can also be represented as

$$f = d \odot (\alpha_1 + \dot{x}_1) \odot (\alpha_2 + \dot{x}_2) \odot \dots \odot (\alpha_{2^n-1} + \dot{x}_1 + \dot{x}_2 + \dots + \dot{x}_n)$$

where $d = 1$ when D is odd and $d = 0$ when D is even.

3.3 Minimization Problems

From the discussion of Section 2.2.5, it is obvious that a realization in the form of Figure 2.6(a) or (b) is directly obtainable from the Reed-Muller canonic forms if the collector row consists of either

Exclusive Or or Equivalence gates. Unfortunately, the resulting realizations may not be minimal.

It would be desirable to have a procedure similar to that used in theunate function case for finding a minimal number of cascades needed to realize a given function. However, there is a unique covering problem associated with functions expressed in the Reed-Muller canonic forms as will be discussed later in this section. It will be shown that an exact solution to this covering problem is not feasible from a practical point of view. Nevertheless, Reed-Muller canonic forms are of interest. It has long been conjectured that a two level realization of a switching function as a mod 2 sum of products is more economical than the conventional Shannon normal form realizations. But due to the nature of the Exclusive Or and Equivalence operators, there has been very little success so far in the development of minimization algorithms for functions expressed as mod 2 or complement mod 2 sum-of-products (or sums).

In the following sections the various minimization problems will be discussed and minimization algorithms will be developed for functions expressed in the Reed-Muller canonic forms.

Besides providing a solution to a classical minimization problem, it will be shown in Section 3.6 how these results can be applied to the realization of arbitrary switching functions using two-dimensional cellular arrays.

With this in mind, the following minimization problems can be posed with respect to the Reed-Muller canonic forms.

A. Given a switching function $f(x_1, \dots, x_n)$, develop algorithms to obtain mod 2 or complement mod 2 expressions of f having a minimum number of product or sum terms which may contain literals from $X = (x_1, \dots, x_n)$ in both complemented or uncomplemented forms.

B. Given a switching function $f(x_1, \dots, x_n)$, develop algorithms to obtain mod 2 or complement mod 2 expressions of f having minimum number of product or sum terms which may contain literals with a fixed polarity from $X = (x_1, \dots, x_n)$, that is, the variable x_i may occur in the product or sum terms either uncomplemented or complemented, but not both.

Problem A is the most general minimization problem for the mod 2 or complement mod 2 expansion for f and was first considered by Muller.

(30) Unfortunately, no efficient method has yet been developed for functions of variables $n \geq 4$. The problem of obtaining the minimal mod 2 sum of products forms is essentially an "odd-even" covering problem in which a set of selected vertices in the n -cube has to be covered by a minimum number of product terms (analogous to covering by implicants in Quine's method) such that each of the selected vertices has been covered an odd number of times, and the remaining vertices in the n -cube could

be used as "optional" or "don't care" states with the constraint that if any one of them is included in the function it has to be covered by the product terms an even number of times. The problem of obtaining the minimal complement mod 2 sum of products forms can be stated in similar language. The duals of each of the above problems have analogous formulations. Even, Kohavi, and Paz⁽¹⁰⁾ have proposed a simple iterative method for simplification of mod 2 sum-of-product expressions, but their method does not guarantee minimality. They have also obtained certain upper bounds for specific classes of functions. A coding theory formulation of the same problem has been proposed by Kautz⁽²⁰⁾ but its implementation for functions of $n \geq 4$ is very unwieldy.

An alternative coding theory formulation (or equivalently a graph theoretic formulation) of the problem is described in the Appendix to this chapter. Here again the disadvantage is that the problem becomes unmanageable for functions of $n \geq 4$. The problem seems to be an extremely difficult one and needless to say, entirely novel approaches have to be developed before the problem is satisfactorily solved.

Problem B has received very little attention from the switching theorists. To the best knowledge of this author, no reliable work has been reported on this topic. In the following sections systematic minimization algorithms are presented for obtaining minimal mod 2 sum-of-products or complement mod 2 sum-of-sums with the polarities of the in-

put variables fixed both for single-output and multiple-output functions. The problems of obtaining minimal mod 2 sum-of-sum terms or minimal complement mod 2 sum-of-products terms are not basically different from the above problems. Thus, minimal forms for equations (3.9) and (3.6) can be obtained directly from the minimal forms of equations (3.5) and (3.8) with the use of Theorems 3.1 and 3.2. Extensions of the method for functions having "don't care" states will also be discussed.

The essential idea of the method consists of obtaining what will be called the polarity functions for the coefficients a_i , b_i , α_i , or β_i in the canonic expansions which will give the logical conditions in terms of the polarities of the input variables such that the coefficient is 0 or 1. In the next step the best polarities for minimal expressions are determined. This step is formulated in terms of a graph theoretic problem and essentially consists of finding the maximum cliques of a linear non-directed graph, for which efficient algorithms have recently been developed.

3.3.1 The First Problem

Consider the problem of reducing the number of non-zero terms in the generalized Reed-Muller expansion given by equation (3.5). We shall call the vector $\dot{X} = (\dot{x}_1, \dot{x}_2, \dots, \dot{x}_n)$ the polarity vector of f in the expansion. Obviously, $\dot{X}$ can have a total of 2^n values since $\dot{x}_1$ is x_1

or x_i . With respect to a given polarity $\dot{X}$, f can be represented by the binary vector

$$\vec{f} = (a_0, a_1, a_2, \dots, a_{2^n-1}).$$

The problem is to determine the appropriate polarities $\dot{X}$ such that the vector f contains a minimum number of 1's.

We shall first consider a simple two variable function $g = x_1' x_2'$ - a single minterm. The vectors $\vec{g}$ are plotted in a Karnaugh map shown in Figure 3.1 in which the co-ordinates x_1 and x_2 denote the polarities of x_1 and x_2 and the entries are the binary coefficients of equation (3.5).

		$x_1 \rightarrow$		0	1
x_2 $\downarrow$	0	0 0 0 1			0 0 1 1
	1	0 1 0 1			1 1 1 1

Figure 3.1. Plot of $\vec{g} = (a_0, a_1, a_2, a_3)$ with all polarities of x_1 and x_2 .

It is easily seen that the expressions for a_i denoted $a_i(g)$ are

$$a_0(g) = x_1 x_2$$

$$a_1(g) = x_2$$

$$a_2(g) = x_1$$

$$a_3(g) = 1 \quad (3.18)$$

The above set of equations will be called the polarity functions for the function $g = x_1 x_2$. These equations mean that, for example, if $\dot{X} = (x_1, x_2)$, the coefficients a_i in the expansion with respect to $\dot{X} = (x_1, x_2)$ can be obtained by putting $x_1 = 1$ and $x_2 = 0$ in the equations; that is, $a_0(g) = 0$, $a_1(g) = 0$, $a_2(g) = 1$, and $a_3(g) = 1$ so that $x_1 x_2 = x_2 \oplus x_1 x_2$. In general it is easy to see that the polarity functions for the function $p_0 = x_1 x_2 \dots x_n$ are

$$a_0(p_0) = x_1 x_2 \dots x_n$$

$$a_1(p_0) = x_2 x_3 \dots x_n$$

$$a_2(p_0) = x_1 x_3 \dots x_n$$

.

.

.

$$a_n(p_0) = x_1 x_2 \dots x_{n-1}$$

$$a_{n+1}(p_0) = x_3 x_4 \dots x_n$$

.

.

.

$$a_{2^n-2}(p_0) = x_1$$

$$a_{2^n-1}(p_0) = 1 \quad (3.19)$$

Note that the polarity function $a_i(p_0)$ is the product of uncomplemented literals which do not occur with the coefficient a_i in the complement-free Reed-Muller expansion. This set of variables will be denoted as X_i ; thus $X_0 = (x_1, \dots, x_n)$, $X_1 = (x_2, x_3, \dots, x_n)$, $X_2 = (x_1, x_3, \dots, x_n)$, $\dots$, $X_{2^{n-3}} = (x_2)$, $X_{2^{n-2}} = (x_1)$, and $X_{2^{n-1}} = \emptyset$ (the empty set). The set of functions $a_0(p_0), a_1(p_0), \dots, a_{2^{n-1}}(p_0)$ will be denoted by A_0 . What are the polarity functions A_i for $p_i = \dot{x}_1 \dot{x}_2 \dots \dot{x}_n$? The answer is provided by the following theorem.

Theorem 3.3 The polarity functions A_i for $p_i = \dot{x}_1 \dot{x}_2 \dots \dot{x}_n$ are obtained by complementing x_k in A_0 if $\dot{x}_k = x_k$ in p_i and leaving x_k unchanged in A_0 if $\dot{x}_k = x_k$ in p_i .

Proof: Without loss of generality, we can assume $p_i = x_1' x_2' \dots x_k' x_{k+1} x_{k+2} \dots x_n = x_1' x_2' \dots x_k' (x_{k+1}') (x_{k+2}') \dots (x_n)'$

Thus $a_j(p_i)$ equals the product of literals X_j where $X = (x_1, x_2, \dots, x_k, x_{k+1}', x_{k+2}', \dots, x_n')$. Hence the theorem.

As an example, for the function $h = x_1' x_2'$, the polarity functions are

$$a_0(h) = x_1' x_2'$$

$$a_1(h) = x_2'$$

$$a_2(h) = x_1$$

$$a_3(h) = 1$$

The following theorem and Theorem 3.3 enable one to obtain the polarity functions $a_i(f)$ for any arbitrary function $f(x_1, \dots, x_n)$ expressed in Shannon canonic form.

Theorem 3.4 Let f be the sum of distinct minterms $p_j, p_k, \dots, p_m$, that is, $f = p_j + p_k + \dots + p_m$. Then the polarity functions for f are given by

$$a_0(f) = a_0(p_j) \oplus a_0(p_k) \oplus \dots \oplus a_0(p_m)$$

$$a_1(f) = a_1(p_j) \oplus a_1(p_k) \oplus \dots \oplus a_1(p_m)$$

⋮

$$a_{2^n-2}(f) = a_{2^n-2}(p_j) \oplus a_{2^n-2}(p_k) \oplus \dots \oplus a_{2^n-2}(p_m)$$

$$a_{2^n-1}(f) = 0 \text{ if there are even number of minterms in } f.$$

$$= 1 \text{ if there are odd number of minterms in } f.$$

The above theorem is an elementary consequence of the fact that we can write f also as $f = p_j \oplus p_k \oplus \dots \oplus p_m$.

Example 3.1

Let $f(x_1, x_2, x_3) = x_1'x_2'x_3' + x_1'x_2'x_3 + x_1x_2'x_3' + x_1x_2x_3'$. Then,

$$a_0(f) = x_1'x_2'x_3' \oplus x_1'x_2'x_3 \oplus x_1x_2'x_3' \oplus x_1x_2x_3'$$

$$a_1(f) = x_2'x_3' \oplus x_2'x_3 \oplus x_2'x_3' \oplus x_2'x_3 = x_2'x_3' \oplus x_2'x_3$$

$$a_2(f) = x_1'x_3' \oplus x_1'x_3 \oplus x_1x_3' \oplus x_1x_3 = x_1'$$

$$a_3(f) = x_1'x_2' \oplus x_1'x_2 \oplus x_1x_2' \oplus x_1x_2 = x_1$$

$$a_4(f) = x_3' \oplus x_3' \oplus x_3 \oplus x_3 = 1$$

$$a_5(f) = x_2' \oplus x_2' \oplus x_2 \oplus x_2 = 1$$

$$a_6(f) = x_1' \oplus x_1' \oplus x_1' \oplus x_1' = 0$$

$$a_7(f) = 0$$

We shall now derive a set of rules which will enable us to obtain the polarity functions of a function P which is a product of literals of the form

$$P = \dot{x}_{j_1} \dot{x}_{j_2} \dots \dot{x}_{j_k}$$

where $\dot{x}_{j_i} = x_{j_i}$ or x_{j_i}' , $1 \leq k \leq n$, and the set of variables $X^P = (x_{j_1},$

$x_{j_2}, \dots, x_{j_k})$ is a proper subset of $X = (x_1, x_2, \dots, x_n)$. Recall that

X_i is the set of variables which occur in $a_i(p_0)$ where $p_0 = x_1'x_2' \dots x_n'$.

The rules are as follows:

Rule 1: If $X_i \supset X^p$, then $a_i(P) = 0$ if $X_i \neq X$

$$a_i(P) = \dot{x}'_{j_1} \dot{x}'_{j_2} \dots \dot{x}'_{j_k} \quad \text{if } X_i = X.$$

Proof: Let $\sum (x_1, x_2, \dots, x_n)$ and $\oplus \sum (x_1, x_2, \dots, x_n)$ denote the sum and mod 2 sum, respectively, of all possible n literal product terms formed from the set of literals $\dot{x}_1, \dot{x}_2, \dots, \dot{x}_n$ where $\dot{x}_j$ can be either x_j or x'_j . Without loss of generality we can assume X_i contains the literals $x_{j_1}, x_{j_2}, \dots, x_{j_k}, x_{j_{k+1}}, \dots, x_{j_s}$, where $1 \leq s \leq n$. Then

$$P = x_{j_1} x_{j_2} \dots x_{j_k} \left[\sum (x_{j_{k+1}}, \dots, x_{j_s}) \sum (x_{j_{s+1}}, \dots, x_{j_n}) \right]$$

Thus, when $s = n$, that is $X_i = X$, we have

$$\begin{aligned} a_i(P) &= \dot{x}'_{j_1} \dot{x}'_{j_2} \dots \dot{x}'_{j_k} \left[\oplus \sum (x_{j_{k+1}}, \dots, x_{j_s}) \right] \times \\ &\quad \left[\oplus \sum (x_{j_{s+1}}, \dots, x_{j_n}) \right] \\ &= \dot{x}'_{j_1} \dot{x}'_{j_2} \dots \dot{x}'_{j_k} \end{aligned}$$

But when $s < n$, all the product terms $x'_{j_1} x'_{j_2} \dots x'_{j_k} x_{j_{k+1}} \dots x_{j_s}$ will occur an even number of times in $a_i(P)$ and hence $a_i(P) = 0$.

Rule 2: If $X_i \subseteq X^p$, $a_i(P) = 0$

This rule follows from the fact that P contains an even number of minterms and therefore, by Theorem 3.4, $a_i(P)$ becomes the mod

2 sum of an even number of the same product terms formed out of literals from X_i complemented or uncomplemented.

Rule 3: If $X_i \not\supset X^p$ and $X^p \not\supset X_i$, we have to consider different cases:

Case a: $X_i \cap X^p = \emptyset$, the empty set.

Subcase (i): If $X_i \cup X^p = X$, $a_i(P) = 1$

Subcase (ii): If $X_i \cup X^p \neq X$, $a_i(P) = 0$

Case b: $X_i \cap X^p \neq \emptyset$

Subcase (i): If $X_i \cup X^p = X$, $a_i(P) = \dot{x}_{j_q} \dot{x}_{j_p} \dots \dot{x}_{j_r}$

where $X_i \cap X^p = (x_{j_q}, x_{j_p}, \dots, x_{j_r})$

Subcase (ii): If $X_i \cup X^p \neq X$, $a_i(P) = 0$

Proof: Assume first $X_i \cap X^p = \emptyset$, the empty set, and assume X_i

$= (x_{j_{k+1}}, x_{j_{k+2}}, \dots, x_{j_{k+t}})$. If $k+t = n$, that is, $X_i \cup X^p = X$, then

$a_i(P) = \oplus \sum (x_{j_{k+1}}, \dots, x_{j_{k+t}}) = 1$. If $k+t < n$, all the product

terms $\dot{x}_{j_{k+1}} \dot{x}_{j_{k+2}} \dots \dot{x}_{j_{k+t}}$ will occur an even number of times in $a_i(P)$

and hence $a_i(P) = 0$. Then assume $X_i \cap X^p \neq \emptyset$. We may assume, without

loss of generality, $P = \dot{x}_{j_1} \dot{x}_{j_2} \dots \dot{x}_{j_q} \dots \dot{x}_{j_k}$ and $X_i = (x_{j_q}, x_{j_{q+1}}, \dots,$

$x_{j_k}, x_{j_{k+1}}, \dots, x_{j_{k+t}}$ where $1 \leq q \leq k-1$. When $k+t = n$, that is $X_i \cup X^p$

$$= X, a_i(P) = \dot{x}_{j_q}' \dot{x}_{j_{q+1}}' \dots \dot{x}_{j_k}' \left[\oplus \sum (x_{j_{k+1}}, \dots, x_{j_{k+t}}) \right] = \dot{x}_{j_q}' \dot{x}_{j_{q+1}}'$$

$\dots \dot{x}_{j_k}' = \dot{x}_{j_q}' \dot{x}_{j_p}' \dots \dot{x}_{j_r}'$ (according to our notation). When $k + t < n$, all

the product terms $\dot{x}_{j_q}' \dot{x}_{j_{q+1}}' \dots \dot{x}_{j_k}'$ will occur an even number of times in

$a_i(P)$ and hence $a_i(P) = 0$.

Rule 4: $a_{2^n-1}(P) = 0$. This rule follows from the fact that P

contains an even number of minterms.

We shall now state the generalization of Theorem 3.4.

Theorem 3.5: Let $f = P_1 + P_2 + \dots + P_g + \dots + P_k$, where P_g ($1 \leq g \leq k$)

is any arbitrary product term involving n or fewer literals such that

$P_g P_j = 0$ for all g and j ($1 \leq g, j \leq k, g \neq j$), then

$$a_i(f) = a_i(P_1) \oplus a_i(P_2) \oplus \dots \oplus a_i(P_g) \oplus \dots \oplus a_i(P_k).$$

This theorem is also an elementary consequence of the fact that $f = P_1 \oplus$

$$P_2 \oplus \dots \oplus P_g \oplus \dots \oplus P_k.$$

We shall now illustrate the usefulness of Theorem 3.5 and Rules

1-4 for obtaining the polarity equations of a function specified in

non-canonic form, but in a form as stated in Theorem 3.5 which can be easily obtained for any arbitrary function.

Example 3.2 Let the function f be specified as

$$f = x_2' x_3' x_5' + x_1' x_2' x_3' x_4' + x_2' x_3' x_4' x_5' + x_1' x_2' x_4' x_5'$$

Then the polarity functions are

$$a_0 = x_2' x_3' x_5' \oplus x_1' x_2' x_3' x_4' \oplus x_2' x_3' x_4' x_5' \oplus x_1' x_2' x_4' x_5'$$

$$a_1 = 0 \oplus x_2' x_3' x_4' \oplus 0 \oplus x_2' x_4' x_5' = x_2' x_3' x_4' \oplus x_2' x_4' x_5'$$

$$a_2 = x_3' x_5' \oplus x_1' x_3' x_4' \oplus x_3' x_4' x_5' \oplus x_1' x_4' x_5'$$

$$a_3 = x_2' x_5' \oplus x_1' x_2' x_4' \oplus x_2' x_4' x_5' \oplus 0 = x_2' x_5' \oplus x_1' x_2' x_4' \oplus x_2' x_4' x_5'$$

$$a_4 = 0 \oplus x_1' x_2' x_3' \oplus x_2' x_3' x_5' \oplus x_1' x_2' x_5' = x_1' x_2' x_3' \oplus x_2' x_3' x_5' \oplus x_1' x_2' x_5'$$

$$a_5 = x_2' x_3' \oplus 0 \oplus x_2' x_3' x_4' \oplus x_1' x_2' x_4' = x_2' x_3' \oplus x_2' x_3' x_4' \oplus x_1' x_2' x_4'$$

$$a_6 = 0 \oplus x_3' x_4' \oplus 0 \oplus x_4' x_5' = x_3' x_4' \oplus x_4' x_5'$$

$$a_7 = 0 \oplus x_2' x_4' \oplus 0 \oplus 0 = x_2' x_4'$$

$$a_8 = 0 \oplus x_2' x_3' \oplus 0 \oplus x_2' x_5' = x_2' x_3' \oplus x_2' x_5'$$

$$a_9 = 0 \oplus 0 \oplus 0 \oplus x_2' x_4' = x_2' x_4'$$

$$a_{10} = x_5' \oplus x_1' x_4' \oplus x_4' x_5' \oplus 0 = x_5' \oplus x_1' x_4' \oplus x_4' x_5'$$

$$a_{11} = 0 \oplus x_1'x_3' \oplus x_3'x_5' \oplus x_1'x_5' = x_1'x_3' \oplus x_3'x_5' \oplus x_1'x_5'$$

$$a_{12} = x_3' \oplus 0 \oplus x_3'x_4' \oplus x_1'x_4' = x_3' \oplus x_3'x_4' \oplus x_1'x_4'$$

$$a_{13} = 0 \oplus x_1'x_2' \oplus x_2'x_5' \oplus 0 = x_1'x_2' \oplus x_2'x_5'$$

$$a_{14} = x_2' \oplus 0 \oplus x_2'x_4' \oplus 0 = x_2' \oplus x_2'x_4'$$

$$a_{15} = 0 \oplus 0 \oplus x_2'x_3' \oplus x_1'x_2' = x_2'x_3' \oplus x_1'x_2'$$

$$a_{16} = 0 \oplus x_4' \oplus 0 \oplus 0 = x_4'$$

$$a_{17} = 0 \oplus x_3' \oplus 0 \oplus x_5' = x_3' \oplus x_5'$$

$$a_{18} = 0 \oplus 0 \oplus 0 \oplus x_4' = x_4'$$

$$a_{19} = 0 \oplus x_2' \oplus 0 \oplus 0 = x_2'$$

$$a_{20} = 0 \oplus 0 \oplus 0 \oplus 0 = 0$$

$$a_{21} = 0 \oplus 0 \oplus 0 \oplus x_2' = x_2'$$

$$a_{22} = 0 \oplus x_1' \oplus x_5' \oplus 0 = x_1' \oplus x_5'$$

$$a_{23} = 1 \oplus 0 \oplus x_4' \oplus 0 = x_4'$$

$$a_{24} = 0 \oplus 0 \oplus x_3' \oplus x_1' = x_3' \oplus x_1'$$

$$a_{25} = 0 \oplus 0 \oplus x_2' \oplus 0 = x_2'$$

$$a_{26} = 0 \oplus 1 \oplus 0 \oplus 0 = 1$$

$$a_{27} = 0 \oplus 0 \oplus 0 \oplus 0 = 0$$

$$a_{28} = 0 \oplus 0 \oplus 0 \oplus 1 = 1$$

$$a_{29} = 0 \oplus 0 \oplus 0 \oplus 0 = 0$$

$$a_{30} = 0 \oplus 0 \oplus 1 \oplus 0 = 1$$

$$a_{31} = 0 \oplus 0 \oplus 0 \oplus 0 = 0$$

We are now in a position to describe the algorithm for obtaining minimal mod 2 sum-of-products of a function with fixed polarities of input. The algorithm consists of the following steps:

1. Express the given function as $f = P_1 + P_2 + \dots + P_k$ where $P_i P_j = 0$ for all i and j , $i \neq j$.
2. Compute the polarity functions $a_i(P_j)$ for all i, j ($0 \leq i \leq 2^n - 1$, $1 \leq j \leq k$).
3. Compute $a_i(f)$ as mod 2 sum of $a_i(P_1), a_i(P_2), \dots, a_i(P_k)$.
4. Find the polarities X for which a minimum number of the coefficients $a_i(f)$ have value 1.

We have explained in detail all but the last step, 4. We will now show that this step is equivalent to the determination of the maximum cliques (see Chapter 4 for discussion) of a linear non-directed graph.

First, we note that if $a_i(f) = 0$, the i th term in the expansion of f is absent from the minimal form irrespective of the polarities of the inputs. Similarly, if $a_i(f) = 1$, the i th term is present in all the minimal forms of f for all polarities of inputs. Now, let $a_i(f)$ be a non-trivial function and let $(X)_i^1, (X)_i^2, \dots, (X)_i^{s_i}$ be the complete set of assignments of the values to the input variables X which make $a_i(f) = 0$. These assignments could be specified in canonic or non-canonic forms. For example, $a_0(f)$ of our Example 3.1 becomes 0 for input assignments (100), (110), (011), (010) which can be specified non-canonically as (1-0), (01-) where '-' stands for 0 or 1. We shall now draw a polarity-compatibility graph G as follows: G will have a finite set of vertices, one corresponding to each assignment $(X)_i^u$, $0 \leq i \leq 2^n - 1$, $1 \leq u \leq s_i$. Two vertices $(X)_i^u$ and $(X)_j^v$ such that $i \neq j$ will be joined by an edge in G if and only if $(X)_i^u$ and $(X)_j^v$ are compatible; that is, no input variable has values 1(0) for $(X)_i^u$ and 0(1) for $(X)_j^v$. Note that the vertices $(X)_i^1, (X)_i^2, \dots, (X)_i^{s_i}$ with the same subscript i are not joined to each other even if they are compatible. Obviously, since G contains all possible solutions to the polarity functions, a maximum clique of this graph will correspond to an input assignment which will make a maximum number of polarity functions $a_i(f)$ equal to zero (therefore a minimal solution).

Example 3.3 For the polarity functions $a_0(f)$, $a_1(f)$, $a_2(f)$, and $a_3(f)$ for the functions in Example 3.1, we have

$$\begin{array}{lll} (X)_0^1 = 1-0 & (X)_1^1 = -01 & (X)_2^1 = 1-- \\ (X)_0^2 = 01- & (X)_1^2 = -10 & (X)_3^1 = 0-- \end{array}$$

The polarity-compatibility graph is shown in Figure 3.2. There are two maximum cliques of this graph: $[(X)_0^1, (X)_1^2, (X)_2^1]$ and $[(X)_0^2, (X)_1^1, (X)_3^1]$ and the corresponding input assignments are (110) and (010).

For the first assignment we have $a_0(f) = a_1(f) = a_2(f) = 0$ and $a_3(f) = 1$ and for the second assignments $a_0(f) = a_1(f) = a_3(f) = 0$ and $a_2(f) = 1$. The coefficients $a_4(f)$ and $a_5(f)$ are both 1 for all assignments. So, the two minimal forms are

$$\begin{aligned} f &= x_3' \oplus x_1'x_2' \oplus x_1'x_3' \\ &= x_2 \oplus x_1'x_2' \oplus x_1'x_3' \end{aligned}$$

For functions of smaller numbers of variables, it is more convenient to obtain the assignments for which a maximum number of coefficients become 0, directly from a plot of $a_i(f)$ on the Karnaugh Map, as has been done in Figure 3.3 for our running example in which the integer i has been entered into a cell for which $a_i(f) = 1$.

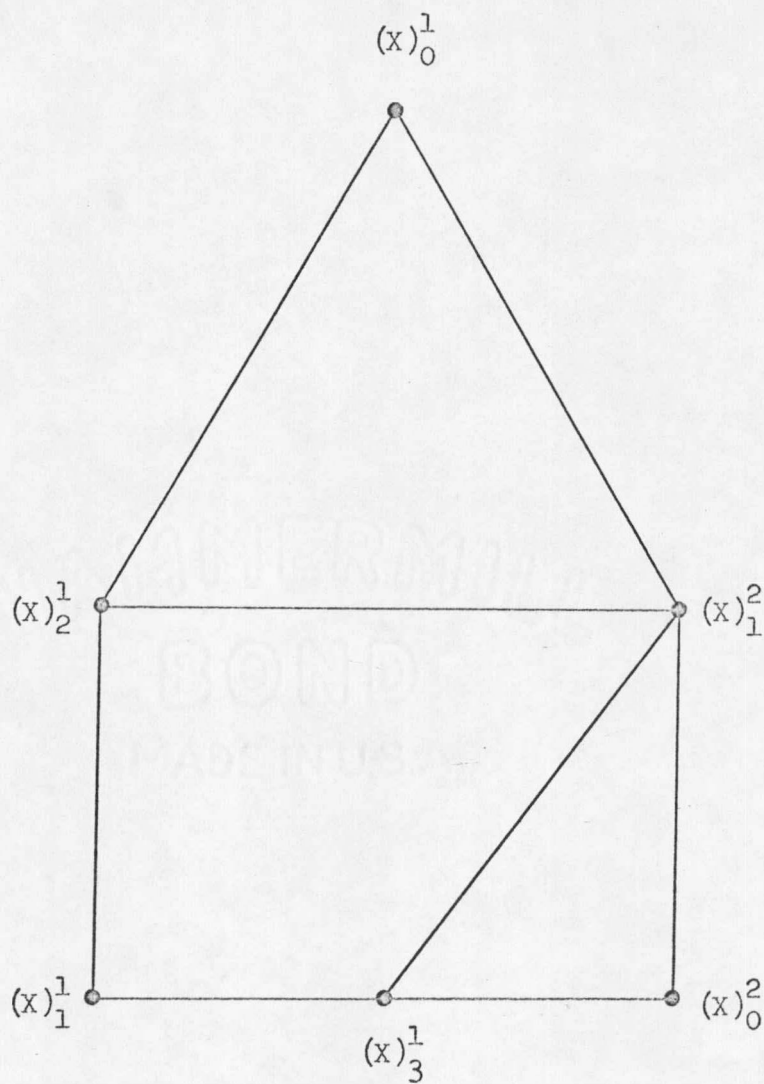


Figure 3.2 The Polarity-Compatibility Graph for Example 3.1.

		x_3		x_2	
		00	01	11	10
x_1	0	012	12	12	2
	1	13	03	013	3

Figure 3.3. Plot of the Polarity Function $a_0(f)$, $a_1(f)$, $a_2(f)$ and $a_3(f)$.

It is readily seen that for assignments 110 and 010, $a_3(f)$ and $a_2(f)$ have value 1, respectively, and the rest are 0, which is the same result obtained before.

As another example, the Karnaugh Map plot of the non-trivial polarity functions for the function of Example 3.2 is shown in Figure 3.4. The coefficients $a_{20}(f)$, $a_{27}(f)$, $a_{29}(f)$, and $a_{31}(f)$ are 1 for all polarities of inputs. The minimal solution is obtained for the polarity assignment $x_1 = 1$, $x_2 = 0$, $x_3 = 0$, $x_4 = 0$, and $x_5 = 1$ and the minimal expression is

$$f = x_3' \oplus x_2'x_3' \oplus x_1x_2'x_5' \oplus x_3'x_4'x_5' \oplus x_1x_2'x_3'x_4' \oplus x_1x_2'x_4'x_5' \oplus x_2'x_3'x_4'x_5'$$

We can now apply Theorem 3.1 to obtain the minimal logical equivalence sum-of-products of the same function as

$x_5 = 0$

$x_3 x_4 \backslash x_1 x_2$	00	01	11	10
00	2,3,6,10,11 12,13,17,18 24,25	0,1,2,4,5,6,8 9,10,11,12,13 15,17,18,19,21 24	1,6,8,9,10 17,18,19,21 22	3,6,10,13,17 18,22,25
01	10,11,13,14 16,17,23,24 25	3,4,7,8,10,11 13,15,16,17,19 21,23,24	7,8,16,17,19 21,22,23	13,14,16,17 22,23,25
11	2,4,5,6,10,11 12,13,14,15, 16,23,25	0,1,2,3,6,7,10 11,12,13,15,16 19,21,23	1,6,7,11,12 16,19,21,22 23,24	4,5,6,11,12 13,14,15,16 22,23,24,25
10	0,3,4,6,10 11,12,13,15 18,25	0,1,5,6,9,10 11,12,13,15 18,19,21	1,2,6,9,10 11,18,19,21 22,24	0,2,3,4,6,10 11,13,15,18 22,24,25

$x_5 = 1$

$x_3 x_4 \backslash x_1 x_2$	00	01	11	10
00	3,10,12,18 22,24,25	5,9,10,12,13 15,18,19,21, 22,24	9,10,18,19 21	3,10,18,25
01	3,14,16,22 23,24,25	3,7,13,15,16 19,21,22,23 24	7,10,16,19 21,23	3,10,14,16 23,25
11	0,3,5,6,11, 12,14,15,16 17,22,23,25	0,1,3,4,6,7,8 11,12,13,15, 16,17,19,21 22,23	1,2,6,7,8,10 12,16,17,19 21,23,24	0,2,3,5,6,10 12,14,15,16 17,23,24,25
10	0,2,3,10,11 12,15,17,18 22,25	2,4,5,8,9,10 11,12,13,15 17,18,19,21 22	2,8,9,10,17 18,19,21,24	0,2,3,10,15 17,18,24,25

Figure 3.4 Plot of the Polarity Functions of Example 3.2.

$$f = 1 \odot x_3 \odot x_2 x_3 \odot x_1 x_2 x_5 \odot x_3 x_4 x_5 \odot x_1 x_2 x_3 x_4 \odot x_1 x_2 x_4 x_5 \\ \odot x_2 x_3 x_4 x_5.$$

3.3.2

The Second Problem

Consider the problem of reducing the number of sum terms in the general complement mod 2 expansion such as

$$f(x_1, x_2, \dots, x_n) = (\alpha_0 + 0) \odot (\alpha_1 + \dot{x}_1) \odot (\alpha_2 + \dot{x}_2) \odot \dots \\ \odot (\alpha_{2^n-1} + \dot{x}_1 + \dot{x}_2 + \dots + \dot{x}_n)$$

The function can be represented by a vector

$$\vec{f} = (\alpha_0, \alpha_1, \alpha_2, \dots, \alpha_{2^n-1})$$

The minimization problem is essentially finding the polarities $\dot{X} = (\dot{x}_1, \dot{x}_2, \dots, \dot{x}_n)$ such that in the representation f there is a minimum number of 0's or maximum number of 1's.

The solution to this problem is analogous to that of the first problem. We first find a set of polarity functions $\alpha_i(f)$ which give logical conditions in terms of the polarities of the input variables under which α_i is 0 or 1. In the second step appropriate input assign-

ments are found for which a minimum number of the $\alpha_i(f)$ functions have value 0.

The rules for obtaining the polarity equations are similar to those discussed before with the difference that we work with 'sums' here rather than 'products'. Thus, for $s_0 = x_1' + x_2' + \dots + x_n'$, the polarity equations are

$$\alpha_0(s_0) = x_1' + x_2' + \dots + x_n'$$

$$\alpha_1(s_0) = x_2' + x_3' + \dots + x_n'$$

⋮

$$\alpha_{2^n-2}(s_0) = x_1'$$

$$\alpha_{2^n-1}(s_0) = 0$$

Theorems analogous to Theorems 3.3 and 3.4 are valid. Thus if f is specified as a product of fundamental sums, $a_i(f)$ is the complement mod 2 sum of the i th polarity functions of the fundamental sums. Similarly, if f is specified as product of sums such that the product of complements of any two sum terms is 0, $a_i(f)$ is then the complement mod 2 sum of the i th polarity functions of the sum terms. The rules for obtaining the polarity functions of a sum term

$$S = x_{j_1}' + x_{j_2}' + \dots + x_{j_k}', \quad k < n$$

are as follows (we denote $X^S = (x_{j_1}, x_{j_2}, \dots, x_{j_k})$ and X_i to be the set of variables which occur in the i th coefficient $\alpha_i(s_0)$, which is the same as X_i before).

Rule 1-A: If $X_i \supset X^S$, $\alpha_i(s) = 1$ if $X_i \neq X$
 $= \dot{x}_{j_1} + \dot{x}_{j_2} + \dots + \dot{x}_{j_k}$ if $X_i = X$

Rule 2-A: If $X_i \subseteq X^S$, $\alpha_i(s) = 1$

Rule 3-A: $X_i \not\supset X^S$ and $X^S \not\supset X_i$

Case a: $X_i \cap X^S = \emptyset$. Then

Subcase (i): $\alpha_i(s) = 0$, if $X_i \cup X^S = X$

Subcase (ii): $\alpha_i(s) = 1$, if $X_i \cup X^S \neq X$

Case b: $X_i \cap X^S \neq \emptyset = (x_{j_q}, x_{j_p}, \dots, x_{j_r})$. Then

Subcase (i): $\alpha_i(s) = \dot{x}_{j_q} + \dot{x}_{j_p} + \dots + \dot{x}_{j_r}$

if $X_i \cup X^S = X$

Subcase (ii): $\alpha_i(s) = 1$ if $X_i \cup X^S \neq X$.

Rule 4-A: $\alpha_{2^n-1}^n(S) = 1$

To obtain the input assignments for which a maximum number of coefficients become 1, we draw a polarity compatibility graph as before, with the difference that the vertices of the graph correspond to the input assignments for which $\alpha_i(f) = 1$. The maximum cliques of this graph will also correspond to minimal solutions.

3.3.3 Functions with Optional States

When the function has a set of optional (don't care) states, the minimization algorithm becomes rather exhaustive or unwieldy. If there are k optional states, all the 2^k possible subsets of these have to be considered together with the given function for determining the polarity functions. Then polarity-compatibility graphs have to be obtained for each set of polarity functions. The minimal solutions will then correspond to those maximum cliques in these graphs whose sizes are the largest amongst all such maximum cliques.

3.3.4 Multiple-Output Functions

The algorithm for obtaining minimal mod 2 or complement mod 2 sum-of-products (or sums) for a set of functions $f_1(x_1, \dots, x_n)$, $f_2(x_1, \dots, x_n), \dots, f_k(x_1, \dots, x_n)$ with a fixed polarity of the input variables is a straight-forward generalization of the single-output case.

Thus, for example, the steps for obtaining minimal mod 2 sum-of-products for $f_1, f_2, \dots, f_k$ can be stated as:

1. Compute the polarity functions $a_i(f_j)$ where $0 \leq i \leq 2^n - 1$ and $1 \leq j \leq k$ by the procedure outlined in the previous sections.
2. Find the polarities X for which a minimum number of the coefficients have value 1 by determining the maximum cliques of the polarity-compatibility graph with respect to all $a_i(f_j)$.

Minimal mod 2 sum-of-sums or complement mod 2 sum-of-products (or sums) for a set of functions $f_1, f_2, \dots, f_k$ can be found by similar extensions of the single-output case.

3.4 Minimal Two-Dimensional Arrays

In this section we shall concentrate on obtaining function realizations based on the minimal length expressions obtained in Section 3.3.1.

Since minimal Reed-Muller forms which may contain literals that appear both complemented and uncomplemented are very difficult to obtain, we will hereafter consider only those realizations based on minimal length expressions which contain literals with a fixed polarity.

Depending on what cell functions are allowed in the two-dimensional cell configurations of Figure 2.6(a) and (b), a minimal size array may or may not be obtained from the minimal canonic forms.

For instance, if the cell functions (other than those in the collector row) are only allowed to be AND(xy), NOR(x' y'), and NIMP(xy, x' y), the realization obtained from the minimal length representation of equation (3.5) will be minimal because one column of the array is needed for each product term appearing in the function expression.

However, if the Exclusive Or operation is also allowed, it may be possible that the minimal length expression does not yield a realization with a minimal number of columns. To illustrate, consider the function

$$f(x_1, x_2, x_3, x_4) = x_1 x_2 \oplus x_1 x_4 \oplus x_2 x_3 \oplus x_2 x_4 \quad (3.20)$$

If the Exclusive Or operator can be used for any cell function, more than one product term can be realized per array column. For instance, the first and second terms can be grouped together as can the third and fourth terms. Then

$$f(x_1, x_2, x_3, x_4) = x_1 (x_2 \oplus x_4) \oplus x_2 (x_3 \oplus x_4) \quad (3.21)$$

which could be realized with an array having only two columns.

Equation (3.21) points out a critical problem in the minimization procedure. Note that the function of equation (3.21) is realizable with an array having two columns (see Section 3.5) only if the variable

ordering is $x_4x_3x_2x_1$. Any other variable ordering would require an array having at least three columns. The question which naturally arises is how to determine (without actually testing) which variable ordering gives the "optimal grouping" such that a minimal number of columns are needed to realize the function.

Unfortunately, the solution to this combinatorial problem becomes unwieldy. Except for very simple functions, all combinations of variable orderings must be tried in order to determine the "optimal grouping".

Because of the exhaustive search needed to find the best variable ordering, a more tractable problem would be to find (for a preset variable ordering) the minimal size array which realizes the minimal length expression. This particular minimization problem is discussed in Section 3.6 after a cascade realizability test applicable to the Reed-Muller canonic forms is developed.

3.5 Cascade Realizability Test for Functions in the Reed-Muller Forms

As in the unate case of Chapter 2, a test and realization algorithm is needed to determine whether a function is realizable by a single cascade of cells. The cell functions will vary depending on the particular canonic expansion in question. In the most general case

where only the uncomplemented input variables are available and the function to be realized has arbitrary input polarity, all the proper functions of two variables may be needed to realize the minimal length expression.

The following algorithm can be used to test functions expressed as in equations (3.5), (3.6), (3.8), and (3.9) for cascade realizability. The algorithm will be described with respect to a function f expressed as in equation (3.5) and can be stated as follows:

Step 1: Check whether f can be expressed in one of the following forms:

$$a) \quad \Phi_1 = f = \dot{x}_{j_1} \dot{x}_{j_2} \dots \dot{x}_{j_k} \Phi_2(X - x_{j_1} - x_{j_2} - \dots - x_{j_k})$$

$$b) \quad \Phi_1 = f = \dot{x}_{j_1} \oplus \dot{x}_{j_2} \oplus \dots \oplus \dot{x}_{j_k}$$

$$\oplus \Phi_2(X - x_{j_1} - x_{j_2} - \dots - x_{j_k})$$

where $X = (x_1, x_2, \dots, x_n)$, x_{j_i} ($1 \leq i \leq k$) is an element of X , $1 \leq k$, and Φ_2 is a function of the variables in X excluding $(x_{j_1}, x_{j_2}, \dots, x_{j_k})$.

Step 2: If f can be expressed in either form a or form b, we have a partial realization of f . The operation of the cell to which the input x_{j_i} is applied is as follows:

- a) If f is in form a, the cell performs the AND operation (xy) or the NIMP operation ($x'y$), depending on whether $\dot{x}_{j_i} = x_{j_i}$ or x_{j_i}' .
- b) If f is in form b, the cell performs an Exclusive Or operation or the Equivalence operation, depending on whether $\dot{x}_{j_i} = x_{j_i}$ or x_{j_i}' .

If f cannot be expressed in either form a or b, f is not realizable in a single cascade.

Step 3: Steps 1 and 2 are repeated iteratively at each step obtaining a Φ_i function from Φ_{i-1} . If at any stage of iteration, a Φ function is obtained which is a two variable function or a function which is just a modulo 2 sum ($\oplus$) or logical product of literals, f is cascade realizable. If not, f is not cascade realizable.

Note that the function f or some intermediate function Φ_i could be of the form $1 \oplus h$ where h is a function of some set of variables. Since $1 \oplus h$ is merely h' , the appearance of a 1 in the Exclusive Or representation means that the operation of the last cell in the cascade must be replaced by its complement. (This is the only time when the NAND, IMP, and OR operations would possibly be needed in realizing equation (3.5)). For example, $1 \oplus x_1 \oplus x_2 \oplus x_3 x_4$ would just become $x_1 \odot x_2 \oplus x_3 x_4$. Or, $1 \oplus x_1 x_2$ would become $x_1 + x_2'$. Similarly, the

presence of a 0 in the Logical Equivalence form can easily be taken care of.

To test equation (3.6) for cascade realizability, the AND operation in form a of Step 1 would be replaced by the OR operation and the AND and NIMP operations of Section a) of Step 2 would be replaced by the OR and IMP operations, respectively. Equations (3.8) and (3.9) can be tested after similar modifications.

The algorithm will be demonstrated with an example in the form of equation (3.5). Let f be represented as

$$f = 1 \oplus x_1 \oplus x_2' x_3' \oplus x_2' x_3' x_4' x_5'$$

$$\text{Step 1: } \phi_1 = f = 1 \oplus x_1 \oplus x_2' x_3' \oplus x_2' x_3' x_4' x_5'$$

$$\text{Step 2: } \phi_2 = x_2' x_3' (1 \oplus x_4' x_5')$$

$$\text{Step 3: } \phi_3 = 1 \oplus x_4' x_5' = x_4 + x_5$$

The cascade which realizes f is shown in Figure 3.5. Note that the cell function associated with x_1 has been changed to Equivalence to account for the presence of the 1 in the expression.

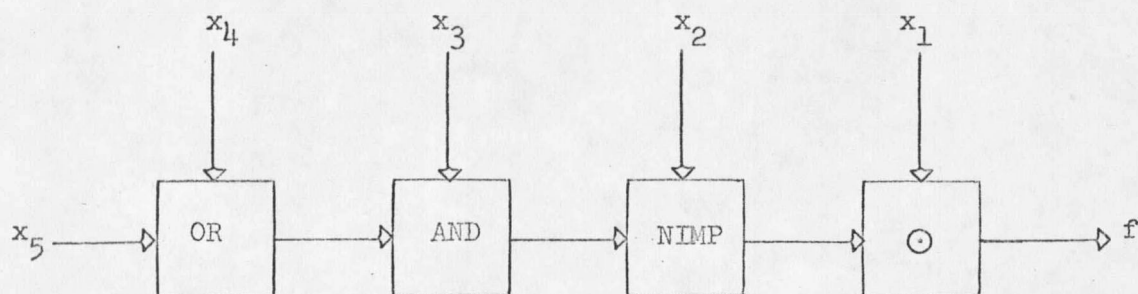


Figure 3.5. Cascade Realization for f .

3.6

Minimal Arrays with Preset Variable Orderings

This section presents an algorithm which finds the minimal size array obtainable from a particular function expression given that the variable ordering has been specified. For example, say we wish to find for variable ordering $x_4x_5x_3x_2x_1$, the minimal size array needed to realize the function

$$f = x_3 \oplus x_2x_3 \oplus x_1x_2x_5 \oplus x_3x_4x_5 \oplus x_1x_2x_3x_4 \oplus x_1x_2x_4x_5 \oplus x_2x_3x_4x_5$$

First, the expression is grouped with respect to the variable x_1 . So

$$f = x_1(x_2x_5 \oplus x_2x_3x_4 \oplus x_2x_4x_5) \oplus x_3 \oplus x_2x_3 \oplus x_3x_4x_5 \oplus x_2x_3x_4x_5$$

Then variable x_2 must be extracted. Now we have

$$f = x_1x_2(x_5 \oplus x_3x_4 \oplus x_4x_5) \oplus x_2(x_3 \oplus x_3x_4x_5) \oplus x_3 \oplus x_3x_4x_5$$

Next, x_3 is extracted, which gives

$$f = x_1 x_2 x_3 (x_4) \oplus x_1 x_2 (x_5 \oplus x_4 x_5) \oplus x_2 x_3 (1 \oplus x_4 x_5) \\ \oplus x_3 (1 \oplus x_4 x_5)$$

After x_5 is extracted, we get

$$f = x_1 x_2 x_3 x_4 \oplus x_1 x_2 x_5 (1 \oplus x_4) \oplus x_2 x_3 (1 \oplus x_4 x_5) \\ \oplus x_3 (1 \oplus x_4 x_5)$$

Each group of terms appearing in the function is realizable using a single cascade. Therefore, four cascades are needed to realize the function if the variable ordering is $x_4 x_5 x_3 x_2 x_1$.

What is needed is an algorithm to extract the variables according to the ordering given and to find the fewest groupings of terms such that each group of terms is realizable in a single cascade. The method centers around the idea of a partition⁽¹⁷⁾ which will now be explained.

Definition: A partition π_i on a set S is a collection of disjoint subsets of S whose set union is S ; i.e.,

$$\pi_i = \{B_\alpha\}$$

such that

$$B_\alpha \cap B_\beta = \emptyset \text{ for } \alpha \neq \beta$$

and

$$\cup \{ B_{\alpha} \} = S$$

We refer to the sets of π_i as blocks of π_i . For example, if $S = \{1, 2, 3, 4, 5\}$, one possible partition would be $\pi_1 = \{ \overline{1}; \overline{2, 3}; \overline{4, 5} \}$, which has three blocks.

We write

$$s \equiv t(\pi_i)$$

if and only if s and t are contained in the same block of π_i . In the example above

$$2 \equiv 3(\pi_1)$$

because 2 and 3 are on the same block of π_1 , but

$$2 \not\equiv 4(\pi_1)$$

because 2 and 4 are in different blocks of π_1 .

If π_1 and π_2 are partitions on S , then $\pi_1 \cdot \pi_2$ is the partition on S such that

$$s \equiv t(\pi_1 \cdot \pi_2) \text{ if and only if } s \equiv t(\pi_1) \text{ and } s \equiv t(\pi_2)$$

The computation of $\pi_1 \cdot \pi_2$ is very simple since the blocks of $\pi_1 \cdot \pi_2$ are obtained by intersecting blocks of π_1 and π_2 . To illustrate, let

$$S = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9 \},$$

$$\pi_1 = \{ \overline{1, 2, 3, 4}; \overline{5, 6}; \overline{7, 8, 9} \} \text{ and } \pi_2 = \{ \overline{1, 6}; \overline{2, 3}; \overline{4, 5}; \overline{7, 8}; \overline{9} \}$$

Then

$$\pi_1 \cdot \pi_2 = \{ \overline{1}; \overline{2, 3}; \overline{4}; \overline{5}; \overline{6}; \overline{7, 8}; \overline{9} \}$$

Consider an n variable switching function $f(x_1, x_2, \dots, x_n)$. Suppose there are m product terms present in the canonic representation of equation (3.5). Let the collection of product terms be denoted by the set S . A variable x_i could be thought to partition the m product terms into two disjoint subsets of S or two subfunctions of f . One subset or subfunction consists of all product terms containing the variable x_i and the other subset or subfunction consists of the remaining product terms. Another variable x_j can further partition the product terms or break the subfunctions up into more subfunctions. (This would correspond to the partition product $\pi_i \cdot \pi_j$).

This partitioning or breaking the original function into subfunctions is precisely the operation which was carried on in the previous example. The grouping of terms corresponds to making a partition about a particular variable. There is a direct correspondence between the number of blocks or subfunctions induced by the partitions and the number of columns needed in the array to realize the function.

To illustrate, we rewrite the function of the previous example.

$$f = x_3 \oplus x_2x_3 \oplus x_1x_2x_5 \oplus x_3x_4x_5 \oplus x_1x_2x_3x_4 \oplus x_1x_2x_4x_5 \oplus x_2x_3x_4x_5$$

Denote product term x_3 as 1, x_2x_3 as 2, and so on. The partitions induced by the variables x_1, x_2, x_3, x_4, x_5 are

$$\pi_1 = \{ \overline{3,5,6}; \overline{1,2,4,7} \}$$

$$\pi_2 = \{ \overline{2,3,5,6,7}; \overline{1,4} \}$$

$$\pi_3 = \{ \overline{1,2,4,5,7}; \overline{3,6} \}$$

$$\pi_4 = \{ \overline{4,5,6,7}; \overline{1,2,3} \}$$

$$\pi_5 = \{ \overline{3,4,6,7}; \overline{1,2,5} \}$$

where

$$S = \{ 1,2,3,4,5,6,7 \}$$

The required variable ordering is $x_4x_5x_3x_2x_1$ so the first partition on S would be

$$\pi_1 = \{ \overline{3,5,6}; \overline{1,2,4,7} \}$$

At this point the function has been broken up into two subfunctions,

$$g_1 = x_1x_2x_5 \oplus x_1x_2x_3x_4 \oplus x_1x_2x_4x_5 \text{ and } g_2 = x_3 \oplus x_2x_3 \oplus x_3x_4x_5 \\ \oplus x_2x_3x_4x_5.$$

If g_1 and g_2 are cascade realizable with the variable ordering given, the process stops and the function can be realized in an array of two columns as $f = g_1 \oplus g_2$. We can test g_1 and g_2 as in Section 3.5 and

we find that neither is realizable with the ordering given.

The next variable to be extracted is x_2 , so the product partition $\pi_1 \cdot \pi_2$ is

$$\pi_1 \cdot \pi_2 = \{ \overline{3,5,6}; \overline{2,7}; \overline{1,4} \}$$

Note that since the x_2 variable was common to all the product terms in g_1 , π_2 did not induce a further partition of $\overline{3,5,6}$. At this point the subfunctions are $g_1 = x_1 x_2 x_5 \oplus x_1 x_2 x_3 x_4 \oplus x_1 x_2 x_4 x_5$, $g_3 = x_2 x_3 \oplus$

$$x_2 x_3 x_4 x_5, \text{ and } g_4 = x_3 \oplus x_3 x_4 x_5.$$

The subfunctions g_3 and g_4 are cascade realizable at this point but g_1 is not. The blocks $\overline{2,7}$ and $\overline{1,4}$ can be deleted from further consideration. Continuing with the block $\overline{3,5,6}$ which corresponds to the function g_1 , we get

$$\pi_1 \cdot \pi_2 \cdot \pi_3 = \{ \overline{5}; \overline{3,6} \}$$

Then $g_5 = x_1 x_2 x_3 x_4$ and $g_6 = x_1 x_2 x_5 \oplus x_1 x_2 x_4 x_5$ which are both cascade realizable with variable ordering $x_4 x_5 x_3 x_2 x_1$. Blocks $\overline{5}$ and $\overline{3,6}$ can now be deleted which accounts for all product terms and therefore the function f can be realized as

$$f = g_3 \oplus g_4 \oplus g_5 \oplus g_6$$

This agrees with the previous result but has been obtained in a very straight-forward manner.

The preceding algorithm can be stated in the following steps:

Step 1: Determine the partitions $\pi_1, \pi_2, \dots, \pi_n$ induced by each of the variables $x_1, x_2, \dots, x_n$. Let S denote the set of product terms.

Step 2: Test to see if the function f is cascade realizable using the algorithm of Section 3.5. If f is realizable we are through; if not, go to Step 3.

Step 3: Partition the set S about the first variable to be extracted as determined by the variable ordering.

Step 4: Test the resulting subfunctions for cascade realizability (Section 3.5).

Step 5: Delete the blocks corresponding to the cascade realizable subfunctions of Step 4. If all partition blocks have been deleted, all the terms in the expression have been accounted for and the process terminates. The function f can now be synthesized using the cascade realizable subfunctions.

If some blocks remain, they are further partitioned about the next variable to be extracted (determined again by the variable ordering). The process continues until all the subfunctions corresponding

to the remaining blocks are cascade realizable with the desired variable ordering.

3.7 A Conjecture

It seems possible that there may exist a function which has a nonminimal length expression whose terms could be grouped in such a manner that the resulting array would have fewer columns than the array produced by the minimal length expression. However, no such example has been found and a conjecture regarding the minimality of the array obtainable from the minimal length expansions of the Reed-Muller canonic forms is made here.

Conjecture: Of all the array realizations obtainable from the Reed-Muller expressions with different input polarities and without any restriction on the variable ordering or cell flexibility, the realization obtained using the minimal length Reed-Muller expression is the one of minimal size.

The conjecture seems intuitively reasonable since there are more terms in the nonminimal length expressions and therefore it is less likely that a better overall grouping could be exhibited amongst these terms. For lack of a counter-example, the conjecture stands.

Chapter 4

CLIQUE DETECTION IN
LINEAR NONDIRECTED GRAPHS

4.1

Introduction

Many of the problems in this thesis were solved in terms of the cliques of a linear nondirected graph. Recall that in Chapter 2, the basic unate components, and in Chapter 3, the minimal length expressions of the Reed-Muller canonic forms, were both found by determining such cliques.

It seems that the cliques of a graph occur in the graph theoretic formulations of many other practical problems as well. For example, cliques find application in switching theory in the simplification of incompletely specified flow tables⁽¹³⁾; in sociology and psychology in the analysis of sociometric data⁽¹⁴⁾; in numerical taxonomy⁽¹⁴⁾; in information retrieval⁽³²⁾; and in the analysis of communication networks⁽¹⁶⁾. Because of the wide application both in this thesis and in other fields and because of the possible exponential growth rate of cliques in a graph⁽⁹⁾, algorithms to enumerate cliques are very important.

This chapter develops a new clique detection algorithm based on a previous work and compares it with other algorithms which have been programmed for this purpose. In order to clearly illustrate these algorithms, a few definitions are needed.

4.2

Definitions

A graph $G = (X, U)$ is defined by a nonempty finite set $X = (x_1, x_2, \dots, x_n)$ of vertices and a finite set $U = (u_1, u_2, \dots, u_m)$ of arcs, together with a relationship which identifies each arc with an ordered pair (not necessarily distinct) of vertices which the arc is said to join. For an arc $u = (a, b)$, the vertex a is called the initial vertex, and b its terminal vertex.

A subgraph of G is a graph $G' = (X', U')$ where $X' \subseteq X$ and $U' \subseteq U$, such that U' contains only those arcs of G which join pairs of vertices in X' . A graph is said to be complete if every pair of vertices is connected in at least one of the two possible directions. A maximal complete subgraph of a graph G is a complete graph which is not a subgraph of any other complete subgraph of G . A clique is a maximal subgraph in which every vertex is connected to every other vertex of the subgraph by an arc. Note that if the graph is symmetric, i.e., $(x_i, x_j) \in U$ whenever $(x_j, x_i) \in U$, then the maximal complete subgraphs are identical to the cliques.

The square matrix $A = [a_{ij}]$ with n rows and n columns is called the connection matrix of the graph $G = (X, U)$ under the convention that $a_{ij} = 1$ ($i \neq j$) if the vertex x_i is connected to the vertex x_j by an arc; otherwise $a_{ij} = 0$. For $i = j$, $a_{ii} = 0$. If A is symmetric, it can be interpreted as the connection matrix of a nondirected graph which has

been obtained from G by replacing each pair of oppositely directed arcs connecting a pair of vertices by a single undirected edge. For a non-directed graph, a clique is defined to be a maximal subgraph whose every pair of vertices is joined by an edge. Since a clique is completely specified by listing the vertices in it, $(v_1, v_2, \dots, v_j)$ will denote the clique containing vertices $v_1, v_2, \dots, v_j$.

Finally, a linear nondirected graph is a nondirected graph which has no self loops and no pair of vertices is connected by more than one edge. A maximum clique is a clique containing a maximum number of vertices.

To clarify some of these ideas, consider the linear nondirected graph of Figure 4.1.

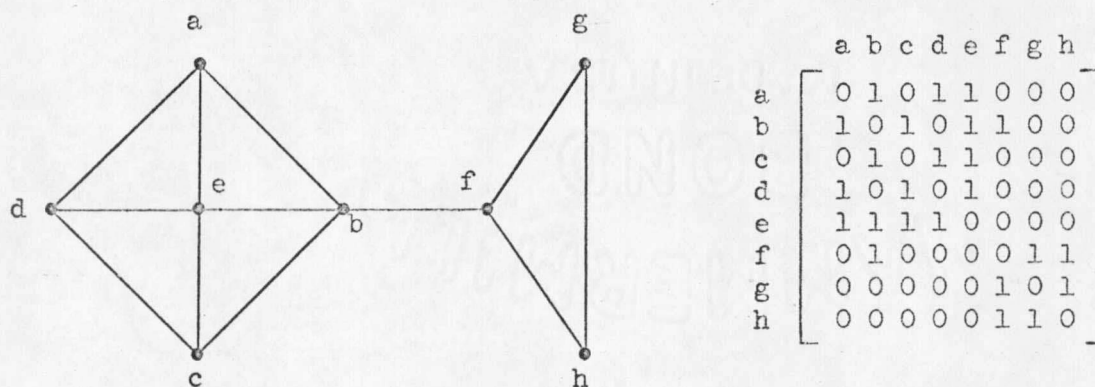


Figure 4.1 Linear Nondirected Graph and Associated Connection Matrix.

In Figure 4.1, the sets of vertices $\{a,e,b\}$, $\{e,b\}$, $\{g,h\}$, $\{d,e,c\}$, etc., belong to subgraphs in which every vertex is connected to every other vertex. However, (a,e,b) , (b,c,e) , (d,e,c) , (a,e,d) , (b,f) , and (f,g,h) are the only cliques of the graph. The subgraphs containing the sets of vertices $\{e,b\}$ and $\{g,h\}$, for instance, are not cliques because they are contained in larger subgraphs which are cliques. The cliques (a,e,b) , (b,c,e) , (d,e,c) , (a,e,d) , and (f,g,h) are all maximum cliques because they all contain the maximum number of vertices of any clique in the graph.

4.3 Algorithms

The first method presented is an improved version of an algebraic formulation of the problem. For the sake of completeness, two other methods from the literature are described and compared with the first algorithm.

4.3.1 Algebraic Formulation

Let us associate with each vertex x_i a binary variable x_i and represent by the Boolean product $x_1 x_j \dots x_k$ the clique formed by the vertices $x_1, x_j, \dots, x_k$. The problem is then to find the Boolean product terms in the variables $(x_1, x_2, \dots, x_n)$ such that each of the terms corresponds to a clique of the graph.

Suppose that vertices x_i and x_j are not joined by an edge in G . Then a product term representing a clique cannot contain the variables x_i and x_j . Therefore, the function

$$F = \sum x_i x_j \text{ (for all distinct } i \text{ and } j \text{ such that } x_i \text{ and } x_j \text{ are not joined by an edge)}$$

represents a complete logical description of the conditions for non-existence of the cliques. Hence the complementary function F' gives the conditions under which the cliques exist. Note that F' is in the form of a product of sums of complemented variables. If F' is expanded and simplified by applying the identity $x' + x'y' = x'$, each product term will be a prime implicant of F' . The product of the uncomplemented variables whose complements do not occur in a prime implicant of F' will, therefore, correspond to a clique of the graph.

The procedure is illustrated with respect to the graph of Figure 4.1. The function F , which represents pairs of vertices not joined in the graph, is given by

$$F = ac + af + ag + ah + bd + bg + bh + cf + cg + ch + df + dg + dh + ef + eg + eh$$

Therefore, F' is given by

$$F' = (a' + c')(a' + f')(a' + g')(a' + h')(b' + d')(b' + g')(b' + h')$$

$$(c' + f')(c' + g')(c' + h')(d' + f')(d' + g')(d' + h') \\ (e' + f')(e' + g')(e' + h').$$

Now, to determine the cliques, F' must be multiplied out and simplified using the identity $x' + x'y' = x'$. The simplification operation in which terms like $x'y'$ are deleted is called the subsume operation; $x'y'$ is the subsumed term. Even for this graph with only eight vertices, the multiplication and simplification involves a fair amount of work.

The formulation presented up to this point is identical to that found in a previous work.⁽⁴⁾ An improvement in this method has been found and it amounts to breaking the original problem up into a set of much smaller problems.

Consider finding only those cliques which contain the vertex x_i . This could be done by constructing a graph containing only x_i and those vertices connected to x_i by an edge (called a neighbor of x_i). The functions F and F' could then be evaluated as before and the cliques containing vertex x_i determined. If the process is repeated for each vertex, all the cliques could eventually be found.

It will now be shown that all the cliques of the graph can be determined by finding only those cliques associated with a particular subset of the vertices in the graph.

Let the neighborhood set of a vertex x_i consist of the vertex x_i and its neighbors (denoted by $[x_i, x_r, x_s, \dots, x_t]$ where $x_r, x_s, \dots, x_t$ are the neighbors of x_i). Then for Figure 4.1, we would have the following neighborhood sets:

$$\begin{array}{ll} [\underline{a}, b, d, e] & [\underline{e}, a, b, c, d] \checkmark \\ [\underline{b}, a, c, e, f] \checkmark & [\underline{f}, b, g, h] \checkmark \\ [\underline{c}, b, d, e] & [\underline{g}, f, h] \\ [\underline{d}, a, c, e] & [\underline{h}, f, g] \end{array}$$

A neighborhood set is said to be maximal if it is not a proper subset of any other neighborhood set. The checked neighborhood sets $[\underline{b}, a, c, e, f]$, $[\underline{e}, a, b, c, d]$, and $[\underline{f}, b, g, h]$ correspond to the maximal neighborhood sets of the graph in Figure 4.1. These sets can easily be determined from the connection matrix of Figure 4.1 if the convention is made that all the diagonal elements are set equal to one. Rows not included in any other row then correspond to the maximal neighborhood sets.

Now let us calculate the cliques corresponding to one of the maximal neighborhood sets, say $[\underline{e}, a, b, c, d]$. The portion of the original connection matrix determined from rows a, b, c, d, e and columns a, b, c, d, e is shown in Figure 4.2.

	a	b	c	d	e
a	1	1	0	1	1
b	1	1	1	0	1
c	0	1	1	1	1
d	1	0	1	1	1
e	1	1	1	1	1

Figure 4.2 Connection Matrix for $[e, a, b, c, d]$

The function F (function describing conditions for nonexistence of cliques) can be determined by inspection from the connection matrix since a 0 means the corresponding vertices are not connected by an edge. So, from Figure 4.2,

$$F = ac + bd$$

Then

$$F' = (a' + c')(b' + d').$$

Note that F contains only two product terms compared to seventeen in the original formulation. This produces a major saving in time and storage as far as actual implementation is concerned.

If F' is expanded we get,

$$F' = a'b' + a'd' + c'b' + c'd'.$$

Therefore, the cliques generated by $[e, a, b, c, d]$ are (c, d, e) , (b, c, e) , (a, d, e) , and (a, b, e) . If we were to calculate the cliques corresponding to one of the nonmaximal neighborhood sets included in $[e, a, b, c, d]$, say $[a, b, d, e]$, we would find that they would all be included in the set generated by $[e, a, b, c, d]$. Therefore, the cliques generated by nonmaximal neighborhood sets are redundant and need not be considered.

If the cliques are calculated for the other maximal neighborhood sets of Figure 4.1, we get (b, f) , (b, c, e) , (a, b, e) , and (f, g, h) . Note that cliques (b, c, e) and (a, b, e) were obtained more than once. At the end of the algorithm all repeated cliques must be deleted. This problem as well as the implied subsume operation in the simplification of F' will be discussed after all the clique detection algorithms have been presented since all the algorithms require a subsume operation in one form or another.

4.3.2 Method of Harary-Ross

The Harary-Ross method⁽¹⁵⁾ proceeds by identifying vertices, called unicliqual, which belong to only one clique. As each unicliqual vertex is identified, its clique is easily determined. The graph is then reduced by removal of all the unicliqual vertices of that clique, together with their associated edges. If at any stage there is no uni-

cliquial vertex, the graph is partitioned and the method applied to each subgraph. All cliques are ultimately obtained.

The following symbols refer to the flow chart of Figure 4.3.

$A =$ adjacency matrix of the nondirected graph G
(diagonal elements are zero).

$M = [m_{ij}]$; m_{ij} is the number of vertices common between the neighborhood sets of v_i and v_j (excluding v_i and v_j) if v_i is joined to v_j ; otherwise $m_{ij} = 0$.

$r(P) =$ row sum for the vertex P in M

$n(P) =$ row sum for P in $A \equiv$ number of vertices cocliquial with P .

$C_P =$ the clique to which P belongs $\equiv P$ and all vertices whose entry in P 's row in A is not zero.

$C_P' =$ the unicliquial vertices of $C_P \equiv$ vertices in C_P whose row sum in A equals that of P .

$G(P) =$ subgraph with P plus all vertices cocliquial with $P \equiv$ all vertices whose entry in the P 's row in M is not zero.

$G(-P) =$ subgraph with vertices in cliques not containing

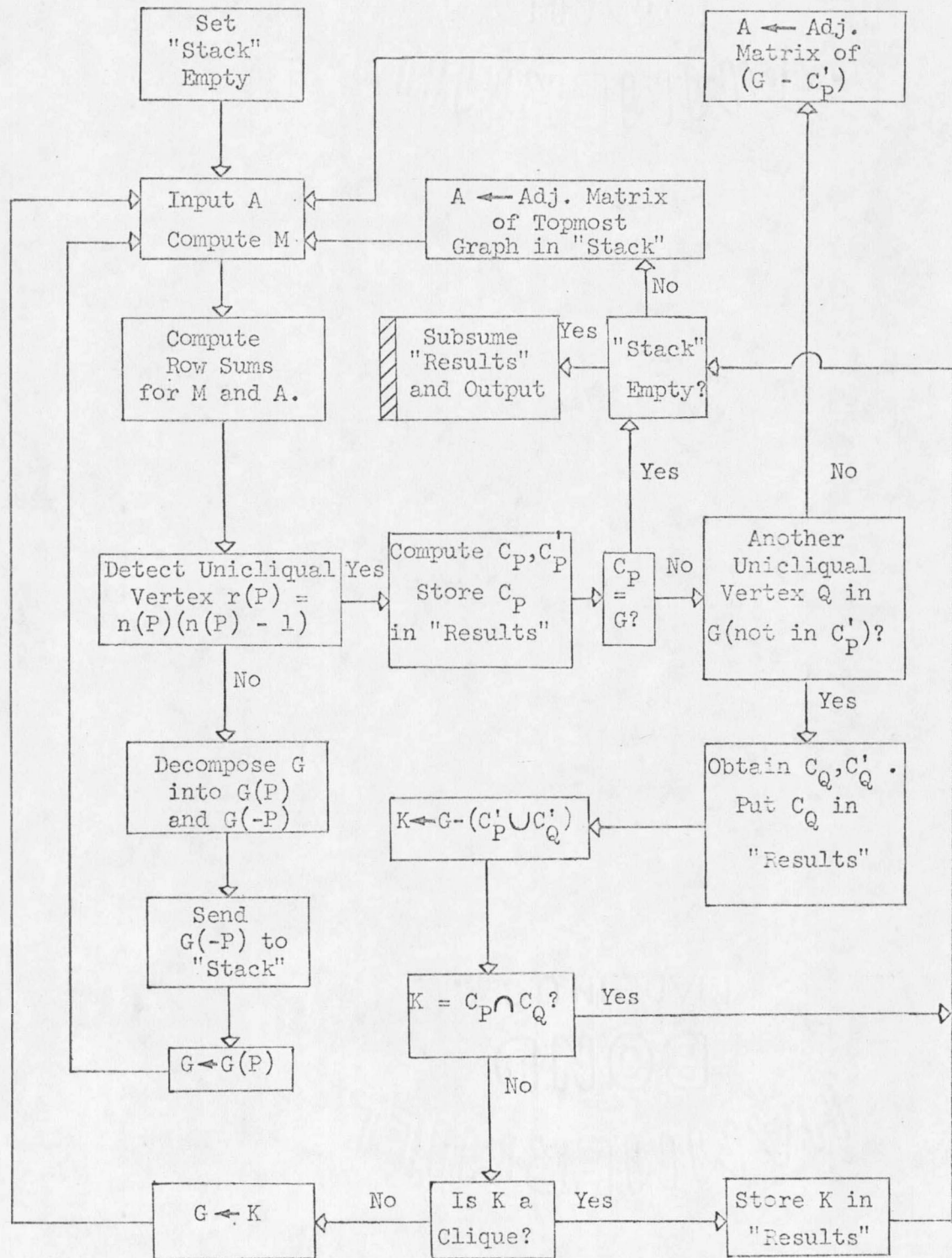


Figure 4.3 Flow Chart of Harary-Ross Method.

$$P \equiv \sum C_R' \text{ where } R \text{ does not occur in } C_P'.$$

The definition of C_P' as it originally appeared in Reference 15 was found to be incorrect. ⁽⁴⁾ The definition of C_P has also been found to be in error and both of the corrected versions appear here.

The method is demonstrated with the graph of Figure 4.1.

	a b c d e f g h	n(P)		a b c d e f g h	r(P)
A =	a [0 1 0 1 1 0 0 0]	3	M =	a [0 1 0 1 2 0 0 0]	4
	b [1 0 1 0 1 1 0 0]	4		b [1 0 1 0 2 0 0 0]	4
	c [0 1 0 1 1 0 0 0]	3		c [0 1 0 1 2 0 0 0]	4
	d [1 0 1 0 1 0 0 0]	3		d [1 0 1 0 2 0 0 0]	4
	e [1 1 1 1 0 0 0 0]	4		e [2 2 2 2 0 0 0 0]	8
	f [0 1 0 0 0 0 1 1]	3		f [0 0 0 0 0 0 1 1]	2
	g [0 0 0 0 0 1 0 1]	2*		g [0 0 0 0 0 1 0 1]	2*
	h [0 0 0 0 0 1 1 0]	2*		h [0 0 0 0 0 1 1 0]	2*

Both g and h are uncliquel vertices. Pick g.

$$C_g = (g, f, h) \quad C_g' = (g, h) \quad \text{Store } C_g = (g, f, h)$$

$$A = \text{adjacency matrix of } (G - C_g').$$

	a	b	c	d	e	f	n(P)		a	b	c	d	e	f	r(P)	
A =	a	0	1	0	1	1	0	3	a	0	1	0	1	2	0	4
	b	1	0	1	0	1	1	4	b	1	0	1	0	2	0	4
	c	0	1	0	1	1	0	3	M = c	0	1	0	1	2	0	4
	d	1	0	1	0	1	0	3	d	1	0	1	0	2	0	4
	e	1	1	1	1	0	0	4	e	2	2	2	2	0	0	8
	f	0	1	0	0	0	0	1*	f	0	0	0	0	0	0	0*

$$C_f = (f, b) \quad C_f' = (f)$$

$$\text{Store } C_f = (b, f)$$

$$A = \text{adjacency matrix of } (G - C_g' - C_f')$$

	a	b	c	d	e	n(P)		a	b	c	d	e	r(P)	
A =	a	0	1	0	1	1	3	a	0	1	0	1	2	4
	b	1	0	1	0	1	3	b	1	0	1	0	2	4
	c	0	1	0	1	1	3	M = c	0	1	0	1	2	4
	d	1	0	1	0	1	3	d	1	0	1	0	2	4
	e	1	1	1	1	0	4	e	2	2	2	2	0	8

None of the remaining vertices are uniclqual so the graph must be decomposed. Decompose about vertex a.

$$G(a) = \text{subgraph with vertices } a, b, d, e$$

$$G(-a) = \text{subgraph with vertices } b, c, d, e$$

	a	b	d	e	n(P)	
A =	a	0	1	1	1	3
	b	1	0	0	1	2*
	d	1	0	0	1	2*
	e	1	1	1	0	3

		a	b	d	e	r(P)
M =	a	$\begin{bmatrix} 0 & 1 & 1 & 2 \end{bmatrix}$				4
	b	1	0	0	1	2*
	d	1	0	0	1	2*
	e	2	1	1	0	4

$$C_b = (b, a, e) \quad C_b' = (b) \quad \text{Store } (a, b, e)$$

$$C_Q = C_d = (d, a, e) \quad \text{Store } (d, a, e)$$

If $G(-a)$ is computed it would yield (b, c, e) and (c, d, e) . Therefore, the cliques for the entire graph are (f, g, h) , (b, f) , (a, b, e) , (a, d, e) , (b, c, e) , and (c, d, e) .

4.3.3 Even's Method

The Even algorithm⁽⁹⁾ constructs the cliques by starting with an arbitrary vertex. The next (arbitrary) vertex is checked to see if it could combine with the first vertex to form a possible clique. Each succeeding vertex is tested in a similar manner with the set of possible cliques already formed. After adding each successive vertex, a subsume operation is performed which deletes sets of vertices which will not correspond to a clique. The algorithm takes as many iterations as there are vertices.

The following definitions apply to the algorithm steps given below:

$N(v)$ = neighborhood set of the vertex v . Recall that the neighborhood set for vertex v_i is determined from the i th row of the connection matrix.

S_i = the set of cliques $G_1, G_2, \dots, G_s$ of the subgraph of G containing $v_1, v_2, \dots, v_i$ (the vertices are numbered arbitrarily).

$$C_{i+1} = \left[\begin{aligned} &(G_{s+1} = v_{i+1} \cup (N(v_{i+1}) \cap G_1)), \\ &(G_{s+2} = v_{i+1} \cup (N(v_{i+1}) \cap G_2)), \dots, \\ &(G_{2s} = v_{i+1} \cup (N(v_{i+1}) \cap G_s)) \end{aligned} \right]$$

$$S'_{i+1} = [S_i \cup C_{i+1}]$$

S_{i+1} = the set S'_{i+1} from which non-maximal sets have been eliminated.

The steps of the algorithm for $n > 2$ are

1. Set $S_1 = V_1$ and $i = 1$.
2. Compute C_{i+1}

3. Compute S'_{i+1}

4. Compute S_{i+1}

5. If $i+1 = n$, stop; otherwise, increment i by 1 and go back to Step 2.

The algorithm will be demonstrated using our running example of Figure 4.1. (Let the vertices $(a,b,\dots,h)$ of Figure 4.1 correspond to the vertices $(v_1,v_2,\dots,v_8)$ for simplicity in notation. The connection matrix of Figure 4.1 is repeated here.

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
$v_1 = a$	0	1	0	1	1	0	0	0
$v_2 = b$	1	0	1	0	1	1	0	0
$v_3 = c$	0	1	0	1	1	0	0	0
$A = v_4 = d$	1	0	1	0	1	0	0	0
$v_5 = e$	1	1	1	1	0	0	0	0
$v_6 = f$	0	1	0	0	0	0	1	1
$v_7 = g$	0	0	0	0	0	1	0	1
$v_8 = h$	0	0	0	0	0	1	1	0

Start with vertex $a = v_1 \equiv 1$. Then

$$s_1 = [1]$$

$$c_2 = [(1,2)]$$

$$s_2' = [(1,2), (1)]$$

$$s_2 = [(1,2)]$$

$$c_3 = [(3,2)]$$

$$s_3' = s_3 = [(1,2), (3,2)]$$

$$c_4 = [(4,1), (4,3)]$$

$$s_4' = s_4 = [(1,2), (2,3), (1,4), (3,4)]$$

$$c_5 = [(1,2,5), (2,3,5), (1,4,5), (3,4,5)]$$

$$s_5' = [(1,2,5), (2,3,5), (1,4,5), (3,4,5), (1,2), (2,3), \\ (1,4), (3,4)]$$

$$s_5 = [(1,2,5), (2,3,5), (1,4,5), (3,4,5)]$$

$$c_6 = [(6,2)]$$

$$s_6' = s_6 = [(1,2,5), (2,3,5), (1,4,5), (3,4,5), (2,6)]$$

$$c_7 = [(6,7)]$$

$$s_7' = s_7 = [(1,2,5), (2,3,5), (1,4,5), (3,4,5), (2,6), (6,7)]$$

$$c_8 = [(6,8), (6,7,8)]$$

$$s_8' = [(1,2,5), (2,3,5), (1,4,5), (3,4,5), (2,6), (6,7), \\ (6,8), (6,7,8)]$$

$$s_8 = [(1,2,5), (2,3,5), (1,4,5), (3,4,5), (2,6), (6,7,8)]$$

$$s_8 = [(a,b,e), (b,c,e), (a,d,e), (c,d,e), (b,f), (f,g,h)]$$

4.4

Comparison of Methods

Of the three methods given, the computational steps in the algebraic formulation and in Even's method seem to be the most straight-forward. Depending on the exact nature of the graph under consideration, the method of Harary-Ross could require a great number of iterations. For instance, if the graph does not contain any uncliqual vertices, it is partitioned into two subgraphs, each of which may contain almost as many vertices as the original graph. The process is repeated until each of the subgraphs are decomposed far enough to yield uncliqual vertices. If many partitions are needed, the "stack" in the algorithm contains many subgraphs, each of which has its own M and A matrices and associated computations. If, on the other hand, the graph in question contains many uncliqual vertices, the graph decomposes quickly and the Harary-Ross method may be more efficient than the others..

One computation which must be performed in each algorithm is the subsume or deletion operation. In the algebraic formulation, the subsume operation must be performed on the F expression to delete terms logically covered by other terms. In both the algebraic formulation and the method of Harary-Ross, redundant cliques must be deleted at the end of the algorithm. In Even's method, a subsume operation is performed on the sets of vertices each time a new vertex is added to the existing sets. This deletes sets which do not correspond to maximal subgraphs.

The algebraic formulation is particularly well suited for implementation using a language like APL. Both the algebraic formulation and the Harary-Ross method have been programmed in APL and the algebraic method seems to be faster, especially for graphs with many edges. Even's method has also been programmed but not in the same language so comparative execution times, although similar to those of the algebraic method, mean very little. A better way to compare these methods would be to program all of them in an assembly language like SDS's Symbol. Even then, the algebraic method would probably have a distinct advantage because of the existence of the logical operations in Symbol.

Chapter 5

SUMMARY AND RECOMMENDATIONS FOR FUTURE RESEARCH

5.1

Summary

The purpose of this research has been the development of minimization techniques applicable to cellular arrays. In particular, two-dimensional arrays of two-input one-output combinatorial cells were considered in which the interconnection structure was constrained to be "nearest neighbor". The difficulties in the general minimization problems were discussed and minimization algorithms were developed for some important subcases.

In Chapter 2, an algorithm was presented which obtained a realization for an arbitrary switching function using a minimal number of unate cascades. Extensions of the method to functions with optional states and multiple-output functions were also included. In Chapter 3, the Reed-Muller canonic expansions of a switching function were considered and realizations based upon these expressions were found.

Apart from the application to the minimization problems, the following theoretical results were obtained:

- 1) For unate functions, the number of equivalence classes of unate cascade realizable functions is determined, a necessary condition for unate cascade realizability is established, and an improved test for determining unate cascade realizability is given.

- 2) A solution to a classical minimization problem involving the Reed-Muller canonic forms is provided.

3) A new algorithm for detecting the cliques of a linear non-directed graph is given.

5.2 Suggestions for Future Research

5.2.1 The Variable Ordering Problem

The minimization algorithms developed in Chapters 2 and 3 assumed that the variable ordering had been predetermined or preset by other considerations. Finding the "best" ordering required an exhaustive search of all the permutations of the input variables.

A problem which could be more extensively investigated would be to develop a test which would determine (without an exhaustive search) which variable orderings resulted in minimal size arrays for the particular cell configuration in question.

In the unate function case, it seems possible that such a test could be developed because of the structure inherent in the unate functions. In the Reed-Muller canonic expression realizations of Chapter 3, however, development of such a test would be difficult because of the arbitrariness involved in grouping of terms in the expressions.

The ordering problem could be eliminated altogether and better realizations obtained if the array was capable of arbitrarily permuting the input variables in each column. For example, consider the function

$f = \sum (0,1,4,5,6,10,13,14) = wyz' + xyz' + xy'z + wy'$. If the variable ordering is arbitrarily specified to be, say $zyxw$, the resulting unate cascade realization would require four unate cascades. One of the "best" orderings, say $wxyz$, would require an array with three unate cascades. But, if the order of the input variables is allowed to be different in each column, an array with only two unate cascades could be used to realize the function. This is demonstrated in Figure 5.1 where the input variables have been permuted by the P-network.

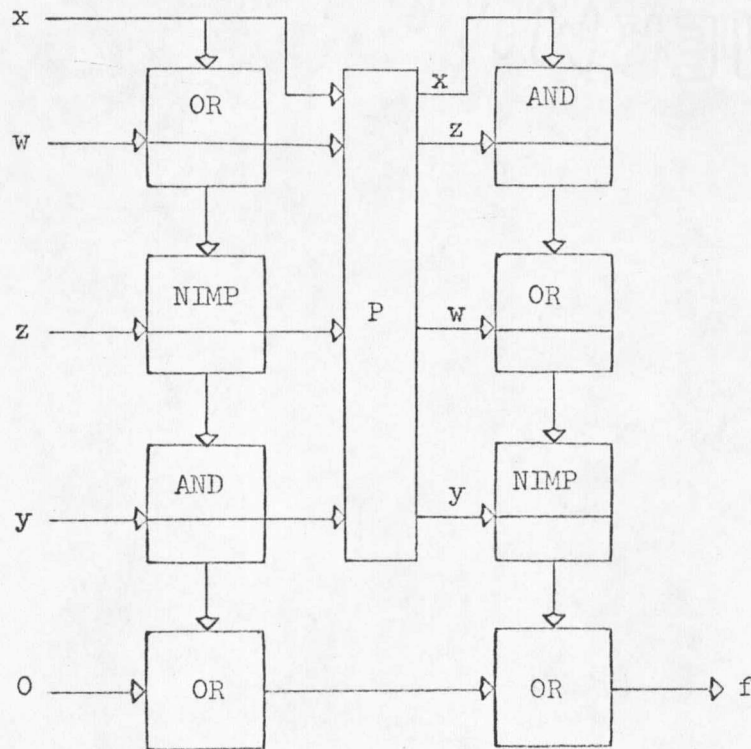


Figure 5.1 Array with Permuter Network.

The complexity of the permuter network would depend on the particular application intended. If the array is needed to realize only one particular function, the permuter network could just consist of fixed connections between cascades. However, if the array is going to be used to realize a large number of functions, the permuter networks would have to be capable of arbitrarily permuting the input variables in each column.

Depending on how many columns are allowed in the array of Figure 5.1, it could also be used as a universal function array. For instance, if the cascade functions in the array are constrained to be unate, a worst case realization for any n variable function would require 2^{n-1} columns. Although there are functions (the odd-even parity function) which require this many columns, it has been the experience of logical designers that functions of this nature are more economically realized using the Exclusive Or operator.

So, despite the fact that no firm upper bound has been established, it is felt that if the OR and Exclusive Or operators were allowed in the collector row of Figure 5.1, the majority of the n variable functions could be realized using much less than 2^{n-1} columns.

5.2.2 Decomposition Approach

The most fundamental concept of switching theory is that of the switching function. Its algebraic properties provide the

foundations for present day logical design procedures. However, the algebraic properties of a switching function may not be the most basic. In the general theory of functions, a more basic property is that of decomposability, the way it can be broken up into a number of simpler functions.

If a function $f(x_1, x_2, \dots, x_n)$ can be decomposed into two or more subfunctions, say $f = \Phi(g_1, g_2, \dots, g_s)$, such that none of the variables $x_1, x_2, \dots, x_n$ are common between the subfunctions $g_1, g_2, \dots, g_s$, the decomposition is said to be disjunctive. Otherwise, the decomposition is nondisjunctive.

To see how decomposition theory may be applied to the problem of obtaining minimal size realizations, consider the array of cells as shown in Figure 5.2.

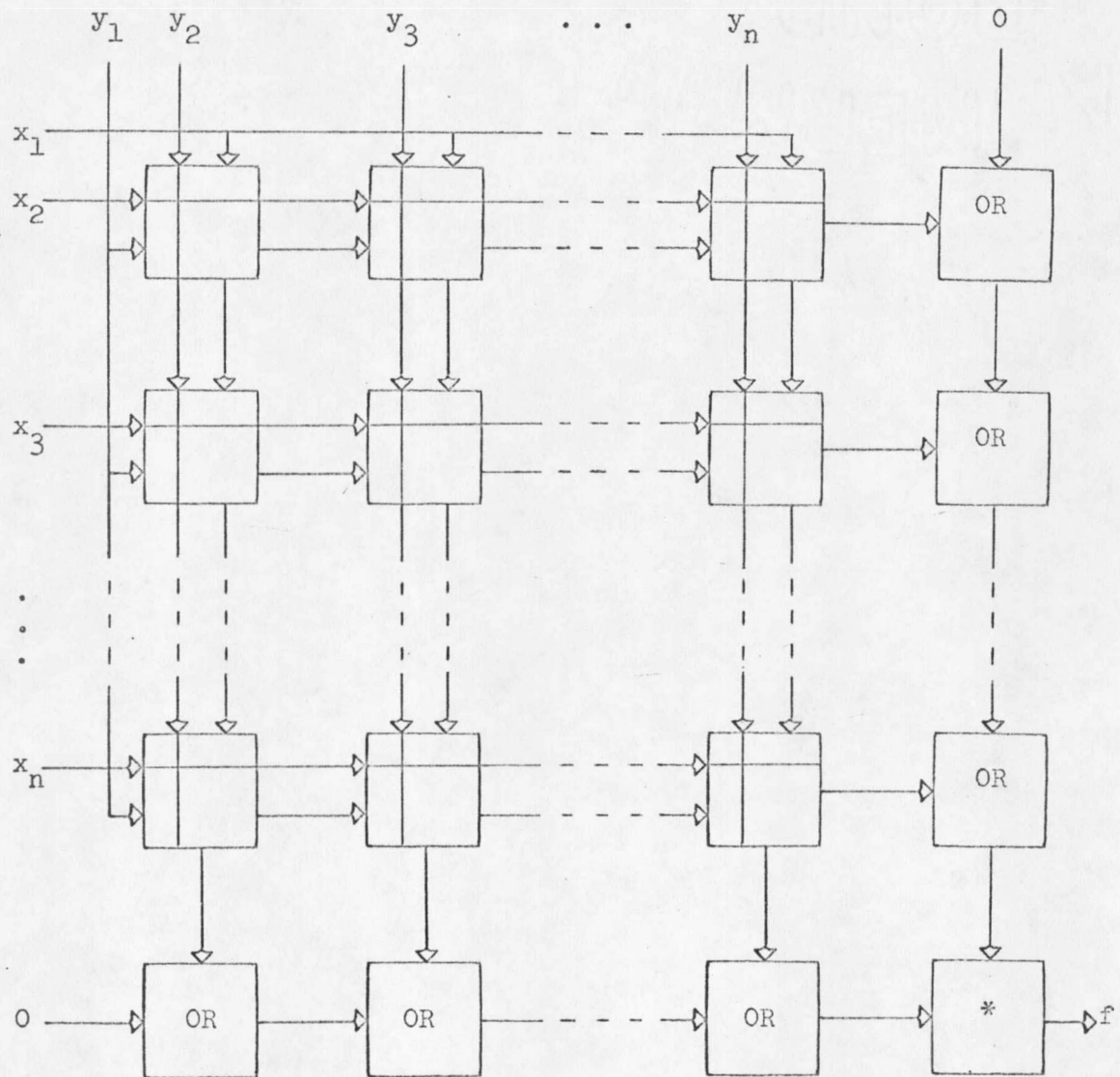


Figure 5.2 Doubly Complex Array.

Each cell has four inputs and two outputs. Input variables are bussed both horizontally and vertically. The horizontal output of each cell is a function of its horizontal input and the vertical bus while the vertical output is a function of the vertical input and horizontal bus. There is a collector row in the horizontal and vertical direction and a single cell joining the two collector rows. Note that permuter cascades can also be introduced in each row and column just as in Figure 5.1.

Suppose that we have a function f which can be disjunctively decomposed as

$$f = g_1 * g_2$$

where $*$ denotes any two variable function. The functions g_1 and g_2 could be separately realized using the techniques developed in Chapters 2 and 3, one in the vertical and one in the horizontal direction in the doubly complex array of Figure 5.2. The cell joining the two collector rows could perform the $*$ operation in $f = g_1 * g_2$ and we have a realization for f based on its decomposition into the simpler functions g_1 and g_2 . Note that the variables of g_1 and g_2 do not necessarily have to be disjoint.

The problem now is to find the proper decomposition of f into two simpler functions such that the resulting array of Figure 5.2 is of minimal size.

There are procedures, some of which are not exhaustive, for testing a function to determine whether it has a decomposition. (1)(6)(29) However, once the decompositions are found, there is no way of determining, except by exhaustion, which particular decompositions yield minimal realizations.

One good solution would be to find a disjunctive decomposition of f into two functions such that half the variables are used to realize each function. If it were possible to find a decomposition of this nature, a worst case realization would only require an array with $(2^{n/2} - 1)$ rows and columns.

It is hoped that further research into the area of function decomposition would provide the necessary results to make the array structure of Figure 5.2 practical. For lack of better methods, the primeunate components of Chapter 2 could be systematically divided and half of them realized in each of the horizontal and vertical directions. The array of Figure 5.2 is quite flexible and other variations, like halving the terms in the Reed-Muller canonic expansions of Chapter 3, are possible as well.

5.2.3 Iterative Decomposition

There is another approach, similar to that described in Section 5.2.2, which could be used in the development of minimal reali-

zations. If the given function f is not realizable in a single cascade, it is decomposed as

$$f = g_1 * f_1$$

where g_1 is a maximal cascade realizable function contained in f and $*$ could be any two variable function. The residue function f_1 is then tested for cascade realizability. If f_1 is not cascade realizable, it will be decomposed as

$$f_1 = g_2 * f_2$$

where g_2 is a maximal cascade realizable function contained in f_1 . The process continues until the residue function f_i is eventually a cascade realizable function. Then

$$f = g_1 * g_2 * \dots * g_{i-1} * f_i$$

Because of the flexibility of the $*$ operator, the iterative approach described here could lead to some very general results.

-137-

APPENDIX

Consider reducing the number of nonzero terms in a general modified ring sum expansion, such as, (for $n = 2$),

$$\begin{aligned} f(x_1, x_2) = & a_{00} \oplus a_{01}x_1 \oplus a_{02}x_1' \oplus a_{10}x_2 \oplus a_{11}x_2x_1 \oplus a_{12}x_2x_1' \\ & \oplus a_{20}x_2' \oplus a_{21}x_2'x_1 \oplus a_{22}x_2'x_1' \end{aligned} \quad (1)$$

where in general there are 3^n terms.

The function $f(x_1, x_2)$ can also be written as

$$f(x_1, x_2) = b_0 \oplus b_1x_1 \oplus b_2x_2 \oplus b_3x_2x_1 \quad (2)$$

where there are 2^n terms and all literals appear unprimed. If each primed literal x_i' in equation (1) is replaced by $(1 \oplus x_i)$, the resulting function expanded, and the coefficients of the x_i 's in equations (1) and (2) equated, we get

$$\begin{aligned} b_0 &= a_{00} \oplus a_{02} \oplus a_{20} \oplus a_{22} \\ b_1 &= a_{01} \oplus a_{02} \oplus a_{21} \oplus a_{22} \\ b_2 &= a_{10} \oplus a_{12} \oplus a_{20} \oplus a_{22} \\ b_3 &= a_{11} \oplus a_{12} \oplus a_{21} \oplus a_{22} \end{aligned} \quad (3)$$

Written in matrix form we get

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_{00} \\ a_{01} \\ a_{02} \\ a_{10} \\ a_{11} \\ a_{12} \\ a_{20} \\ a_{21} \\ a_{22} \end{bmatrix} = A \begin{bmatrix} a_{00} \\ a_{01} \\ a_{02} \\ a_{10} \\ a_{11} \\ a_{12} \\ a_{20} \\ a_{21} \\ a_{22} \end{bmatrix} \quad (4)$$

where A is a binary coefficient matrix and the implied addition operation is mod 2.

If we rewrite these equations as

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} b_0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ b_1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ b_2 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ b_3 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ a_{00} \\ a_{01} \\ a_{02} \\ a_{10} \\ a_{11} \\ a_{12} \\ a_{20} \\ a_{21} \\ a_{22} \end{bmatrix} = B \begin{bmatrix} 1 \\ a_{00} \\ a_{01} \\ a_{02} \\ a_{10} \\ a_{11} \\ a_{12} \\ a_{20} \\ a_{21} \\ a_{22} \end{bmatrix} \quad (5)$$

where B is the matrix A (equation (4)) augmented by column $(b_0, b_1, \dots, b_{2^n-1})$, the solution set to these equations is a parity check code of length 3^n+1 .

The code words defined by equation (5) having a 1 in the first component and a minimum number of 1's in the remaining components cor-

respond to the minimum solutions of the modified ring sum expansion.

In general, there is no way of directly enumerating the code words of minimum weight (minimum number of ones). However, if this coding theory problem is solved, the minimization of the modified ring sum expansion as presented here will likewise be solved. In lieu of a coding theory solution, a graph theoretic formulation is presented.

To do this, construct a weighted graph with 2^{2^n} vertices, one for each possible binary vector $(b_0, b_1, \dots, b_{2^n-1})$. Vertex i is connected to vertex j , weighted by one of the column vectors of matrix A , if the mod 2 sum of this column of matrix A and the binary vector associated with vertex i is equal to the vector associated with vertex j . (Note that the graph is symmetric). When this is done for all vertices, the minimal solution is obtained by finding the shortest path from the vertex $(0, 0, \dots, 0)$ to the vertex $(b_0, b_1, \dots, b_{2^n-1})$. The sequence of weights in this path corresponds to a sequence of columns in the A matrix which in turn determines the coefficients whose values are 1 in equation (1). Since algorithms⁽¹⁹⁾ are available which determine shortest paths without enumerating all possible paths, such a formulation may be worthwhile.

Obviously, the graph becomes large for an increasing number of variables. However, it is noted that solutions to all functions of n

variables are represented in the graph. Once the graph has been set up, any other n variable function can be minimized by changing the binary vector $(b_0, b_1, \dots, b_{2^n-1})$.

Kautz⁽²⁰⁾ has a different coding theory formulation of this minimization in which the problem is one of error correction of a certain code. However, as the number of variables increases, the combinational aspect of his formulation seems to become unwieldy, also.

LITERATURE CITED

1. Ashenhurst, R. L., "The Decomposition of Switching Functions," Proc. of an International Symposium on the Theory of Switching, Annals of the Computational Laboratory of Harvard University, Vol. 29, pp. 74-116.
2. Bartee, T. C., "Computer Design of Multiple-Output Logical Networks," IRE Trans. on Electronic Computers, Vol. EC-10, No. 2, pp. 21-30, March 1961.
3. Brown, D. R., A High-Speed Multi-Position Electronic Switch, S. M. Thesis, Course VI, MIT, May 1947.
4. Calingaert, P., and Mukhopadhyay, A., "Programmed Algorithms For Determining Cliques of a Linear Graph." Private Communication.
5. Comfort, W. T., "Highly Parallel Machines," IBM Rep. No. 62-825-496, Oct. 1962.
6. Curtis, H. A., The Design of Switching Circuits, D. Van Nostrand Co., Inc., Princeton, New Jersey, 1962.
7. Elspas, B., "Topological Constraints on Interconnection-Limited Logic," IEEE Special Publication, Switching Circuit Theory and Logical Design, Vol. S-164, pp. 133-197, Oct. 1965.
8. Elspas, B., et. al., Investigation of Propagation Limited Computer Networks, SRI Project 4523, April 1964 and July 1965.
9. Even, S., "An Algorithm for Generating All Maximal Full Subgraphs of a Given Graph," R-68-163 (IEEE Computer Group Repository).
10. Even, S., Kohavi, I., and Paz, A., "On Minimal Modulo 2 Sums of Products for Switching Functions," IEEE Trans. on Electronic Computers, Vol. EC-16, pp. 671-674, October 1967.
11. Forrester, J. W., "Digital Information Storage in Three-Dimensions Using Magnetic Cores," J. Appl. Phys. 22 (Jan. 1951), pp. 44-48.
12. Gonzales, R., "A Multilayer Iterative Circuit Computer," IEEE Trans. on Electronic Computers, Vol. EC-12, pp. 781-790, Dec. 1963.
13. Grasselli, A., and F. Luccio, "A Method For Minimizing the Number of Internal States in Completely Specified Sequential Networks," IEEE Trans. on Electronic Computers, Vol. EC-14, pp. 350-359, June 1965.

14. Harary, F., Norman, R. Z., and Cartwright, D., Structural Models: An Introduction to the Theory of Directed Graphs, New York: Wiley, 1965.
15. Harary, F., and Ross, I. C., "A Procedure for Clique Detection Using the Group Matrix," Sociometry, Vol. 20, pp. 205-215, 1957.
16. Harary, F., and Ross, I. C., "The Number of Complete Cycles in a Communication Network," Journal of Social Psychology, Vol. 40, pp. 329-332, 1956.
17. Hartmanis, J., and Stearns, R. E., Algebraic Structure Theory of Sequential Machines, Englewood Cliffs, New Jersey: Prentice-Hall, 1966.
18. Holland, J. H., "Iterative Circuit Computers," Proc. Eastern Joint Computer Conference, pp. 108-113, 1959.
19. Hu, T. C., "Revised Matrix Algorithm for Shortest Paths," J. Soc. Industrial and Applied Mathematics, Vol. 15, pp. 207-218, 1967.
20. Kautz, W. H., "Concerning the Minimization of Modified-Ring-Sum Expansions," Internal Memorandum of SRI, 13 May, 1965.
21. Maitra, K. K., "Cascaded Switching Networks of Two-Input Flexible Cells," IRE Trans. on Electronic Computers, Vol. EC-11, pp. 136-143, April, 1962.
22. McCluskey, E. J., Introduction to the Theory of Switching Circuits, McGraw-Hill Book Co., New York, 1965.
23. McCluskey, E. J., and H. Schorr, "Essential Multiple-Output Prime Implicants," Mathematical Theory of Automata, Proc. Polytechnic Inst. Brooklyn Symposium, Vol. 12, pp. 437-457, April, 1962.
24. Minnick, R. C., "A Survey of Microcellular Research," J. ACM, Vol. 14, pp. 203-261, April, 1967.
25. Minnick, R. C., "Cobweb Cellular Logic," Proc. of the AFIPS Fall Joint Computer Conference, Vol. 27, Part I, pp. 327-341, 1965.
26. Minnick, R. C., "Cutpoint Cellular Logic," IEEE Trans. on Electronic Computers, Vol. EC-13, pp. 685-698, Dec. 1964.
27. Minnick, R. C., et. al., Cellular Arrays for Logic and Storage, SRI Report, Project 4641, April, 1966.

28. Mukhopadhyay, A., "Unate Cellular Logic," IEEE Trans. on Computers, Vol. C-18, pp. 114-121, February, 1969.
29. Mukhopadhyay, A., et. al., Minimization of Cellular Arrays, Final Report, NSF Contract GJ-158, Electronics Research Laboratory, Endowment and Res. Foundation, Montana State University, Bozeman, Montana, September, 1969.
30. Muller, D. E., "Application of Boolean Algebra to Switching Circuit Design and to Error Detection," IRE Trans. on Electronic Computers, Vol. EC-3, pp. 6-12, September, 1954.
31. Rajchman, J. A., "Static Magnetic Matrix Memory and Switching Circuits," RCA Review, 13 (June 1952), pp. 183-201.
32. Roth, J. P., and Karp, R. M., "Minimization Over a Boolean Graph," IBM Journal of Research and Development, Vol. 6, No. 2, pp. 227-238, April, 1962.
33. Scudder, F. J., and Reynolds, J. N., "Crossbar Dial Telephone Switching System," Trans. AIEE 58 (May 1939), pp. 179-192.
34. Short, R. A., "Two Rail Cellular Cascades," Proc. of the AFIPS Fall Joint Computer Conference, Vol. 27, Part I, pp. 355-369, 1965.
35. Slotnick, D. L., Borck, W. C., and McReynolds, R. C., "The Solomon Computer," Proc. AFIPS Fall Joint Computer Conf., Vol. 22, pp. 97-107, 1962.
36. Unger, S. H., "A Computer Oriented Toward Spatial Problems," Proc. IRE 46, 10 (Oct. 1958), pp. 1744-1750.
37. von Neuman, J., "Theory of Automata: Construction, Reproduction, Homogeneity," Part II of The Theory of Self-Reproducing Automata, ed. A. W. Burks. Univ. of Illinois Press, Urbana, Ill., 1966.
38. Yoeli, M., "A Group Theoretical Approach to Two Rail Cascades," IEEE Trans. on Electronic Computers, Vol. EC-14, pp. 815-821, December, 1965.

MONTANA STATE UNIVERSITY LIBRARIES

3 1762 10011376 8

cop.2 Minimization methods
for cellular logic arrays

NAME AND ADDRESS	DATE	REMARKS
...	...	...

8-17-11

Tim Berry (6/10/09)

D 378

Sch 56

Cap. 2