Fall 1985

High Performance Switching Circuits for VLSI

Ali Reza Feizi
Old Dominion University

HIGH PERFORMANCE SWITCHING CIRCUITS FOR VLSI

by

Ali Reza Feizi
BSEE May 1983, Old Dominion University, Norfolk, VA

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF ENGINEERING

ELECTRICAL ENGINEERING

OLD DOMINION UNIVERSITY
December 1985

Approved by:

S.V. Kanetkar (Director)

John W. Stoughton

S. Zahorian

ACKNOWLEDGEMENTS

The author would like to express gratitude to Dr. Damu Radhakrishnan for his invaluable contributions and insightful guidance in the preparation of this thesis. Thanks also go to the committee members: Dr. S. V. Kanetkar, Dr. John W. Stoughton and Dr. S. Zahorian for their time and constructive criticism.

Finally, the author wishes to thank many of his colleagues in the Electrical Engineering Department for their friendship and assistance.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Recent advances in LSI and VLSI technologies has brought forward many device configurations for IC design that are not equivalent to a simple interconnection of logic gates. The major emphasis on IC design is aimed at high performance, minimum size, and reduced power. Pass networks and CMOS logic have evolved as two candidates best suited in this regard.

A pass network is defined as an interconnection of a set of pass transistors which realize a particular switching function. A unified switching theory is given in [1] to represent pass networks and procedures are developed for the design of any combinational circuit using pass transistors. A special class of pass networks called Binary Tree Structured (BTS) pass networks appear to require the fewest number of transistors to realize pass network functions [2]. BTS pass networks have certain other advantages related to IC design, including fault detection and the design of multiple-output networks.

This thesis considers four different problems presently encountered in the design of pass networks.

Present design of BTS pass networks is based on the use of a Karnaugh map. Algorithmic procedures available for the design of pass networks are not suitable for the generation of a function in BTS form. In addition, the algorithmic procedure currently available is very slow and uses a considerable amount of memory. Therefore, a new algorithm for the design of BTS pass networks is developed in this thesis. This algorithm is more efficient in time as well as the space complexity than all the known algorithms.

Next the design of multiple-output pass networks is considered. It is proved that BTS pass networks are most suitable for the design of multiple-output networks.

Programmable Logic Arrays (PLA) receive wide acceptance by the VLSI design community because of its structural regularity and the ease of design. Any finite state machine can very easily be designed around a PLA. These PLAs can in general be considered as multiple-output networks. Conventional design of a PLA is based on the AND-OR plane concept. A new kind of PLA called pass network PLA is proposed here. It is based on multiple-output pass networks and uses almost zero static power.

Finally, fault diagnostics play a major role in present designs. As the circuits are made smaller and more complex, it becomes very difficult to probe the internal nodes using the limited number of pins available on the chip. Though there are some techniques available for the fault detection

in pass networks [2,4]. The fault detection in multiple-output networks are considered in this thesis and it is found that fault detection in these networks can be carried out using the methods available for the single-output networks. It is also shown that some assumptions made to detect some faults in pass networks may not be satisfied in some cases.

1.1 Thesis Outline

Chapter 2 begins with an introduction to pass networks. Many of the important definitions used in the literature are repeated here. The design procedures given in this chapter are divided into two: one for general pass networks and the other for BTS pass networks.

Chapter 3 develops an algorithmic procedure for the design of BTS pass networks. A data structure for the storage of pass implicants is first introduced followed by specifying the necessary conditions for combining the pass implicants for minimization. The BTS pass network algorithm is divided into two parts: the first part generates the pass prime implicants using the above mentioned data structure and the second part shows the actual generation of the BTS pass function from these prime implicants. This chapter concludes by providing a comparitive study of the new algorithm with the earlier one available for the design of general pass networks.

Chapter 4 is devoted to the synthesis of multiple-output pass networks including both general and BTS pass networks. Necessary and sufficient conditions are derived for sharing parts of the network among different pass functions.

Chapter 5 begins with an introduction to PLAs as used in conventional designs. A stick diagram representation for the conventional PLA structure is given followed by a proposed structure of a pass network PLA. This chapter concludes with the discussion of the different trade-offs involved in selecting a PLA for a particular application.

Fault detection techniques oriented to multiple-output pass networks is the subject of chapter 6. Pass variable faults and control variable faults are treated separately. The necessary and sufficient conditions to be satisfied for detecting stuck-at, stuck-open, and stuck-on faults are derived in this chapter. In addition, conditions under which a test becomes invalid and possible solutions to some of these problems are given in this chapter. Conclusion and additional research is presented in chapter 7.

CHAPTER 2

PASS LOGIC REALIZATION

A pass network is defined as an interconnection of a set of pass transistors to realize a particular switching function. These networks are of three kinds: NMOS, PMOS, and CMOS networks. An NMOS pass network passes a good 0 but a poor 1, and a PMOS pass network passes a good 1 and a poor 0. A CMOS pass network, on the other hand, passes both a good 0 and a good 1. In an NMOS transistor when the Gate voltage is a logic 1 (greater than the threshold voltage), the value at the input is presented to the output. When the gate voltage is a logic 0 (below the threshold voltage), the output is in the high impedance state (open circuit). The same is true for PMOS transistors except for the reversal of the logic inputs applied to the gate.

2.1 Definitions

The signal inputs to the gates of the NMOS transistors are called control signals and the associated variables are called control variables (C). The signals which feed the inputs (source/drain) of NMOS/PMOS transistors are called pass signals and the associated variables are called pass

variables (V). A pass implicant is defined as a product term P_i (product of one or more control variables) passing the variable V_i to the output and is denoted by $P_i(V_i)$. An NMOS pass transistor, its truth table, and its logic representation are shown in Fig. 2.1. Due to the bidirectional nature of MOS transistors, the Source and the Drain leads can be interchanged.

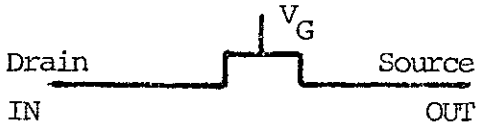
2.2 Generalized Pass Networks

The design procedures for both NMOS and CMOS pass networks are given in [1]. This is repeated here to give the basic understanding of these networks. Only NMOS pass networks are considered further in this thesis.

A generalized pass network is shown in Fig. 2.2. The output pass function F can be expressed as the sum of pass implicants:

$$F = P_1(V_1)+P_2(V_2)+\ldots+P_i(V_i)+\ldots+P_n(V_n).$$

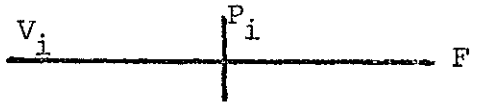
Each pass implicant $P_i(V_i)$ passes the pass variable V_i to the output when the corresponding product term P_i = 1. The product term P_i is the product of a number of variables called control variables, representing the pass transistors which are connected in series. The pass implicant is similar to the traditional implicant; but, instead of representing only the 1's of the function, it represents either the 1's, 0's or a combination of both 1's and 0's.



(a) Electronic Representation.

V_G	OUT
1	IN
0	High Impedance

(b) Truth Table.



$F = P_i (V_i)$

(c) Logic Representation.

Fig. 2.1 An NMOS Pass Transistor.

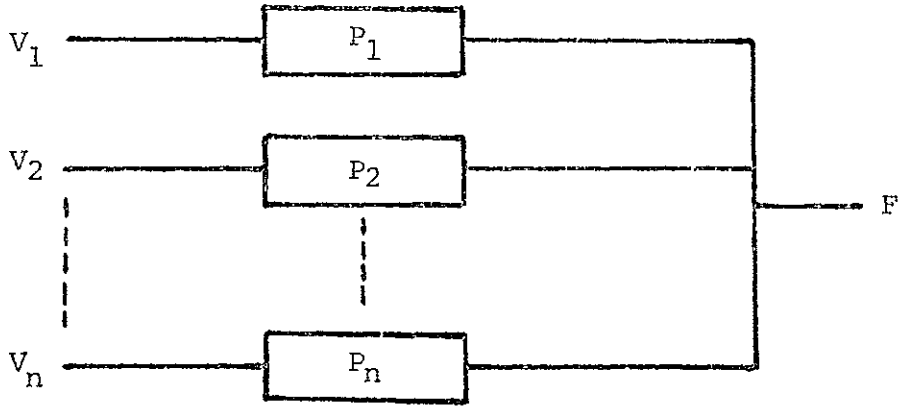


Fig. 2.2. A Pass Network Representation.

2.3 Pass Network Implementation Using Karnaugh-Map

The K-map minimization of a pass function is given in [1]. Since pass transistors are basically tri-state devices, the output of a pass network will be defined only if a path is activated from input to output. The design of a pass network using a K-map involves finding minimal set of pass prime implicants to cover the whole map. These pass implicants and their associated pass variables are identified by the following procedure:

1) A pass implicant consists of 2^i cells in the K-map, each cell in the implicant has i adjacent cells in the implicant. The product term defining the implicant is identical to the traditional implicant.

Each and every entry in the map is covered at least once in the formation of a pass function thus preventing any high impedance condition occurring at the output of the network. Pass implicants, formed in this manner using a K-map, which subsumes no other pass implicants with fewer number of literals are called pass prime implicants. The generation of a pass function using a K-map is shown in Fig. 2.3. The cells forming each pass prime implicant are encircled in Fig. 2.3. The simplified pass function is given by: $F = X_2(X_4') + X_4'(X_2) + X_4X_2'(X_3)$, and is shown in Fig. 2.4.

A conflict condition occurs whenever two pass implicants are enabled simultaneously to the output of a

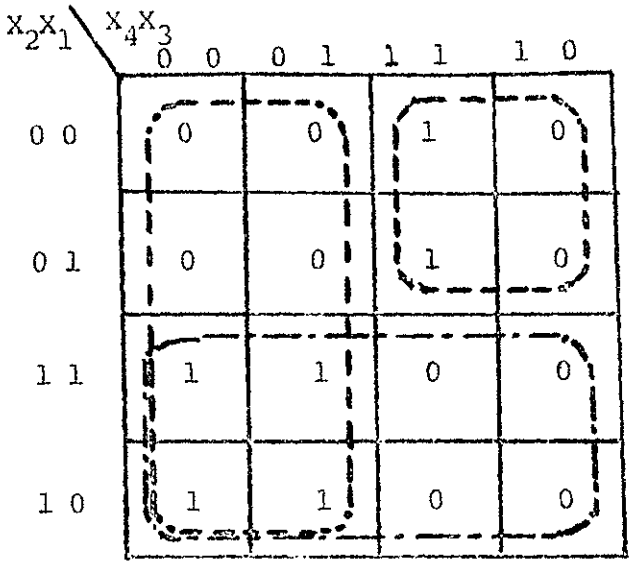


Fig. 2.3. K-Map Minimization of a Pass Function.

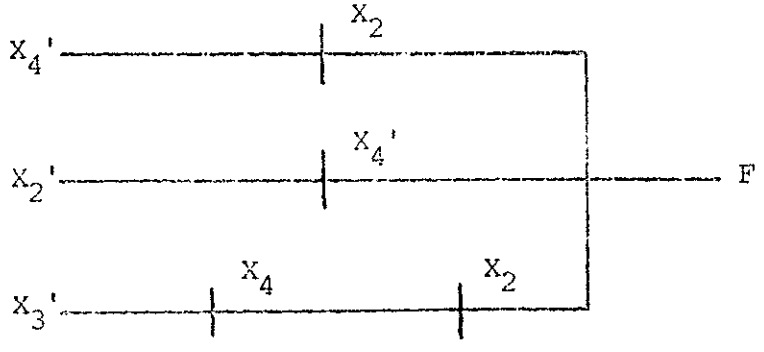


Fig. 2.4. Pass Network for the K-Map of Fig. 2.3.

pass network with opposite pass variable logic values. This causes a short circuit in the pass network which is not desirable. The above design procedure guarantees that such a conflict will not occur if the function is completely specified [1]. Care must be taken in the design of incompletely specified pass functions to prevent such a conflict.

2.4 Binary Tree Structured Pass Networks (BTS)

A pass network designed using the K-map minimization procedure described above may not be optimal in the number of transistors in its realization. This is because the same literal may occur in a number of pass prime implicants. One way to minimize the number of transistors used in the realization is by implementing the function as a BTS pass function.

A BTS pass network is characterized by having exactly two branches at every node, with their control variables complementing each other. Two different approaches for the design of a BTS pass network are given in [2]. A brief review of these two approaches is given below.

2.4.1 Karnaugh-Map Partitioning Approach

There are two goals in partitioning a K-map. The first is to create a BTS pass network. The second is to minimize the number of transistors in the network. The K-map partitioning approach may be described as follows:

1) Choose a control variable, X_i, and form a BTS pass network with two branches and one node. One branch is controlled by X_i, and the other branch is controlled by X_i'. The original map is now partitioned into two segments defined by $X_i=0$ and $X_i=1$.

2) Apply the same procedure to each segment of the map, supplying the output as a pass term to the earlier branch controlled by X_i or X_i'.

3) Repeat step 2 until the pass variable of each branch is in the set $\{0, 1, X_i, X_i'\}$.

The following example illustrates the above procedure:

Example 2.1: Consider the K-map shown in Fig. 2.3. Choose X_2 as the first partition. The first two branches and the root node may be drawn immediately. This is shown in Fig. 2.5. The submaps corresponding to partitions $X_2= 0$ and $X_2=1$ are shown in Fig. 2.6.

Now each submap is dealt with separately. In Fig. 2.6, the output corresponding to $X_2=0$ partition is neither a constant nor an input variable. Choose X_4 as the partitioning variable in this submap and add a node to the previous circuit as shown in Fig. 2.7. The two resulting submaps are shown in Fig. 2.8.

The output of the function (or pass variable) partitioned by $X_2=0$ and $X_4=0$ is 0. Therefore, the input to the branch controlled by X_4' is 0. Similarly, the output of the function partitioned by $X_2 = 0$ and $X_4 = 1$ is X_3. So input to the branch controlled by X_4 is X_3. The output of

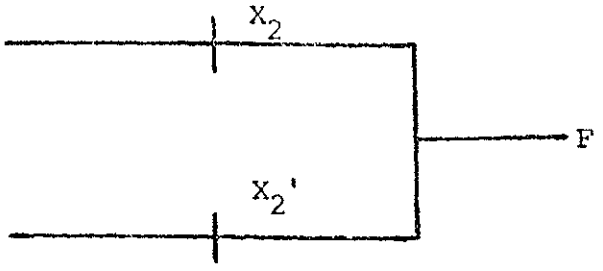


Fig. 2.5. Network for X_2 Partition.

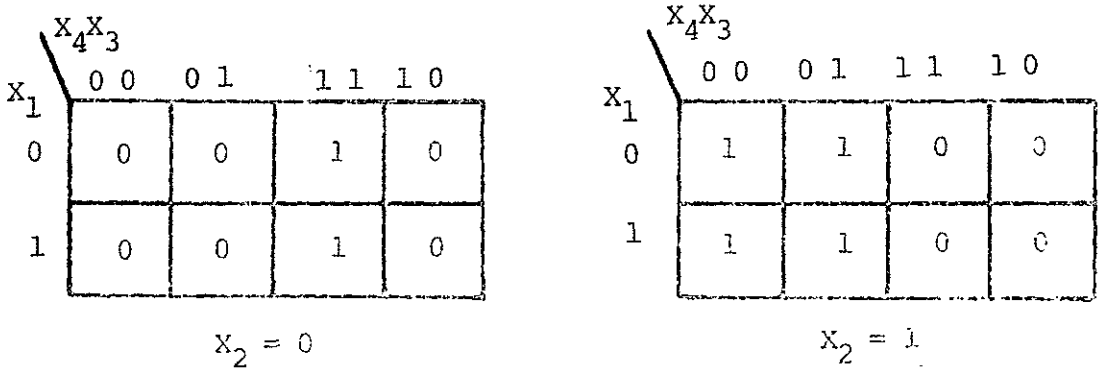


Fig. 2.6. Submaps for X_2 Partition.

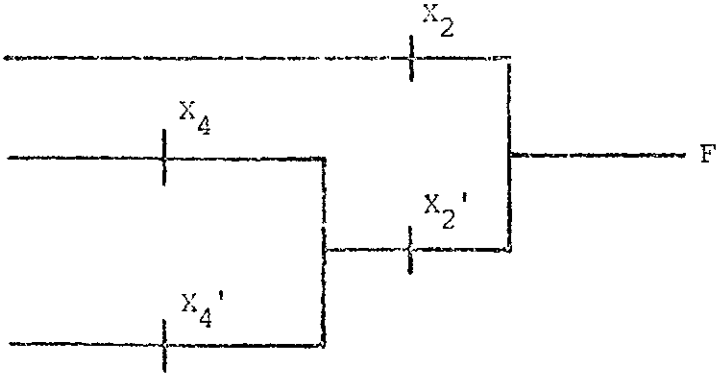


Fig. 2.7. Network for X_2 and X_4 Partitions.

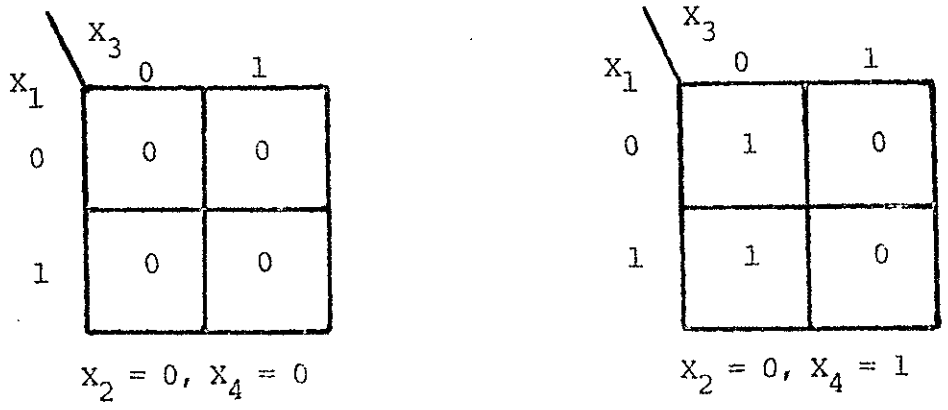


Fig. 2.8. Submaps for $X_4 = 0$, and $X_4 = 1$ with $X_2 = 0$.

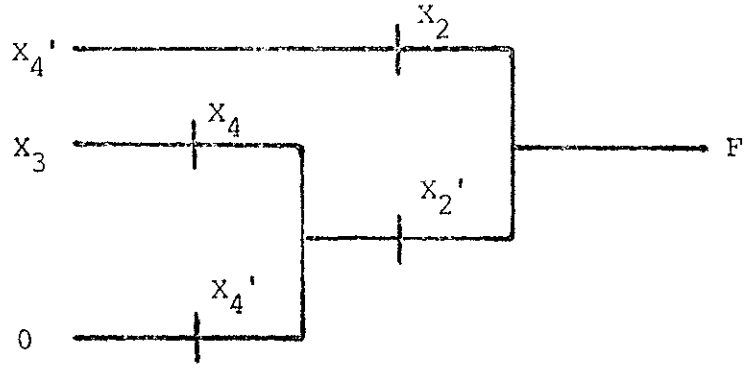


Fig. 2.9. A BTS Pass Network for the K-Map of Fig. 2.3.

the function partitioned by X_2=1 is X_4' and therefore X_4' is the input to the branch controlled by X_2. The complete circuit is shown in Fig.2.9.

For a function of n variable, at most n - 1 partitions are required to generate all the pass implicants. Each pass implicant will at least be a two cell group. Thus, it requires at most 2^n - 2 transistors to implement a function of n variables as a BTS pass function [2].

2.4.2 Factoring Approach

Consider an arbitrary pass function which contains two terms of the form $X_iP_i(V_i)+X_iP_i'(V_j)$. This expression is equivalent to $X_i[P_i(V_i+P_i'(V_j)]$, where X_i is factored out of both the terms. Therefore, if X_i appears in more than one pass implicant, it can be factored out. Consider the following pass function:

$F = X_2X_1(X_4)+X_2'X_1'(X_3)+X_2'X_1(0)+X_2X_1'(1)$, note that there are two choices for factoring this function. Taking X_2 as the factor, we get:

$F = X_2[X_1(X_4)+X_1'(1)]+X_2'[X_1(0)+X_1'(X_3)]$, which is in BTS form. Thus, any pass function is a candidate for factoring as long as the pass implicants do not overlap. In addition, factoring would result in some minimization.

A pass network with disjoint pass implicants is called a disjoint pass network. In a disjoint pass network, when any one product term P_i is equal to 1, all the other terms are 0's. A BTS pass network is a disjoint pass network but the converse need not always be true.

CHAPTER 3

BTS PASS NETWORK DESIGN

The design of a BTS pass network using the K-map minimization procedure given earlier becomes difficult as the number of variables exceed more than five. An algorithmic procedure is given in [1] for the design of the general pass network. However, this algorithm is very slow and uses considerable amount of memory. In addition, the pass prime implicants generated in this manner may have cells in common thus eliminating the generation of BTS pass networks.

In developing an algorithm for BTS pass networks optimality is given only a secondary consideration. The major emphasis given here is memory requirement and speed. Also, the algorihtm can only be used to implement completely specified functions.

3.1 Data Structure

The data structure used in this algorithm is very similar to the one used in [1] with the inclusion of an extra field, called the Repeat field (R-field) which is added to each pass implicant record. Thus a record has four fields: a Base field, a Difference field, a Pass

field, and a Repeat field. The basic structure of a record is shown in Fig. 3.1. The different fields are defined as:

(a) Base field (B) - This is an integer field that represents the cells in the truth table. The base field functions identically to the base field of the traditional tabular method and can be described in any desired base. In the presentation here, base 10 is used. When the pass implicant covers more than one cell, the lowest value will be listed. For example, when two adjacent cells $X_1'X_2X_3'$ and $X_1'X_2X_3$, represented by their decimal equivalents 2 and 3, combine together to form a pass implicant, 2 becomes the base field entry.

(b) Difference field (D) - This is also an integer field, consisting of one or more integer entries separated by commas. The difference field has an identical counterpart in the traditional tabular method. The entries in this field represent the difference between the cell representations of the terms which are combined together to form the pass implicant. The difference field for the implicant formed by the combination of two terms whose decimal equivalents are 2 and 3 is (1), whereas it is (1,4) for an implicant formed of 2(1) and 6(1). The ordering of the entries in the difference field bears no significance; thus (1,4) and (4,1) are equivalent.

(c) Pass field (p) - This is an alphanumeric field which represents the pass variable that is to be passed by the pass implicant. The pass field can contain 0, 1, X_i or X_i'.

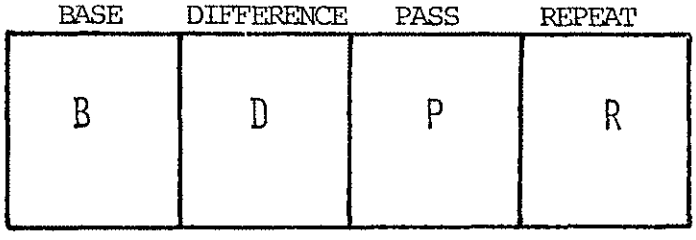


Fig. 3.1. Pass Implicant Record.

(d) Repeat field (R) - This is an integer field, 0 or 1, which determines if the record should be used for further comparisons and possible minimization. A 1 entry means the record should be used for further comparisons and a 0 entry means that the record should be disregarded. Whenever two records are combined together to form a new record, the R-field in the new record is set to 1.

3.2 Pass Implicant Generation

The necessary and sufficient conditions for combining two pass implicant records are:

(a) The B-fields differ in only one binary bit.

(b) The D-fields agree,

(c) The R-fields have logic value 1, and

(d) The P-fields must be a constant or an identical pass variable.

Once all the above conditions are satisfied the pass implicant records are combined using Theorem 1 in [1]. This is illustrated below.

Consider the cells C_i and C_j of the function $f(X_n,\ldots,X_1, X_0)$ with outputs assigned to be f_i and f_j respectively. Let C_i and C_j be adjacent with variable X_k = 0 in C_i and X_k = 1 in C_j. The pass implicant records excluding the R-fields are:

(a) If $f_i=f_j=0$ then the record is C_i (2^k) 0

(b) If $f_i=f_j=1$ then the record is C_i (2^k) 1

(c) If $f_i=0$ and $f_j=1$ then the record is C_i (2^k) X_k

(d) If f_i=1 and f_j=0 then record is C_i (2^k) X_k'

The above conditions suggest that when two records A & B with identical pass fields are combined to form a new record C, the pass field in C will be the same as that of A.

3.3 BTS Pass Network Algorithm

The algorithm uses three linear lists, K, M, and F of pass implicant records. The K-list is used to store the given function. The M-list is used to store the pass implicants generated during the course of the algorithm and the F-list is used to store the final result. The K and M-lists are interchanged during each iteration.

The K-list has 2^n records and M and F have 2^{n-1} records each for an n-variable function. A pointer is attached to each one of these lists (k, m, and f for K, M, and F lists respectively) and it is assumed that whenever a record is written in a list it is written in the location indicated by this pointer. Similarly, whenever a reference is made to a record in a list, it implies the record pointed by the corresponding pointer. The records pointed by the pointers in the K, M, and F lists are denoted by K_k, M_m, and F_f respectively. Before the start of the algorithm, the K-list is initialized with the given function as follows:

The B and P-fields are initialized with the decimal number from 0 to 2^n - 1 and their corresponding output functions respectively. The D and R-fields in all the

records are initialized to 0 and 1 respectively. Also the pointer f is initialized to 0, and N is initialized to n. The flag X = 1 denotes the formation of a new record in the last iteration. The algorithm is divided into two parts:

(i) The generation of pass prime implicants in the BDPR format, and

(ii) The conversion from BDPR format to BTS form.

3.3.1 Pass Prime Implicant Generation

1. If N = 0 then go to 8.
2. If X = 0 then go to 9, else set k=0, m=0, and X=0.
3. If $k=2^N$ then go to 7.
4. Compare the records K_k and K_{k+1} for possible combination. If they combine, the new record formed is stored in M_m, set X=1, and then go to 6, else set the R-field in M_m to 0.
5. If the R-fields in both K_k and K_{k+1} are 1, then transfer K_k and K_{k+1} to F_f and F_{f+1} respectively and increment f by 2, else if the R-fields of both K_k and K_{k+1} are different, then transfer one of K_k or K_{k+1} to F_f depending on whichever has a logic value of 1 in its R-field. Increment f by 1.
6. Increment m by 1, k by 2 and then go to 3.
7. Decrement N by 1, interchange the K and M lists, and then go to 1.
8. If X=1, transfer M_0 to F_0.

9. Halt.

The F-list now contains the result.

3.3.2 Conversion of BDPR Format to BTS Form

The BTS pass prime implicants generated by the algorithm given above are in BDPR form. For describing the conversion from BDPR form to BTS form we need only to interpret the Base and Difference fields properly. Hence in the following only these two fields are used and the pair taken together will be called a BD record.

A given D-field for an n-variable function can in general be written as: $(2^0,2^1,\ldots,2^{n-1})$. Let b represent the B-field entry and d represent the sum of the D-field entries. For a given BD record, first determine the number of entries in the D-field and denote it by Z. The number of variables appearing in the product term is now given by Y = n - Z. A Y-digit binary number, C, can be derived by dividing b by d + 1. This binary number C represents the product term P_i. The MSB of C represents the highest significant variable in the function. (A 1 represents a non- complemented variable and a 0 represents a complemented variable).

Once all the BD records are converted using the above procedure, the function can be expressed as the sum of disjoint pass implicants. In order to arrive at a BTS pass function, the factoring approach described earlier can be used. This factoring must be done in complementary pairs.

First the variable X^* must be factored out from all the terms in which it is present and then the variable $X^{*'}$. The factoring procedure must start with the highest significant variable first and then continue with the lowest significant variables.

Example 3.1: Consider the three variable function:

$F(X_3\ X_2\ X_1) = m(0,3,6,7)$

Table 3.1 illustrates the minimization procedure for the function F. The procedure begins by listing all the 8 pass implicant records in the K-list of Table 3.1(a). The pass field is set to 1 for all the records with B-fields 0, 3, 6, and 7. The two records K_0 and K_1 (0 0 1 1 and 1 0 0 1) are compared for possible combination.(In the first iteration, they always combine). This produces the result 0 1 X_1' 1 which is stored in M_0 (Table 3.1(b)). The procedure is now continued with K_2 and K_3 as the new records. The M-list at the end of this iteration is shown in Table 3.1(b). The contents of the present M-list and the K-list are now interchanged. For illustrative purposes this is done by treating the present M-list as the new K-list; and a new M- list is formed as shown in Table 3.1(c). Comparing K_0 and K_1 (Table 3.1(b)), it is seen that their pass variables do not match and hence cannot be combined. Hence both K_0 and K_1 are sent to the output lists F_0 and F_1 (shown in Table 3.1(d)). This is continued for the remaining records in the K-list (Table

K	B	D	P	R
	0	0	1	1
	1	0	0	1
	2	0	0	1
	3	0	1	1
	4	0	0	1
	5	0	0	1
	6	0	1	1
	7	0	1	1

(a) K-List

M	B	D	P	R
	0	1	x_1'	1
	2	1	x_1	1
	4	1	0	1
	6	1	1	1

(b) M-List at First Iteration.

M	B	D	P	R
				0
	4	1,2	x_2	1

(c) M-List at Second Iteration.

F	B	D	P	R
	0	1	x_1'	1
	2	1	x_1	1
	4	1,2	x_2	1

(d) F-List

Table 3.1. Pass Prime Implicant Generation for Example 3.1.

3.1(d)) in a similar manner. The final result is stored in the F-list (Table 3.1(d)).

As seen the F-list contains 3 records. To illustrate the conversion to BTS form, consider the BD record 4(1,2). The given 3 variable function can be written in BD form as: F = (0, 1, 4). Thus Z = 2, y = 1, b = 4, d = 3, and C = 1. Therefore, the binary number corresponding to the product term is 1 which represents X_3. The pass variable is X_2. This produces the pass implicant $X_3(X_2)$. Similarly the other records are converted. The function can now be expressed as: $F = X_3(X_2) + X_3'X_2(X_1) + X_3'X_2'(X_1')$. In order to convert the function to BTS form, start the factoring process with X_3 and X_3'. The next factoring variable will be X_2 (both X_2 and X_2'). Thus the function F can be written as:

$$F = X_3(X_2)+X_3'[X_2(X_1)+X_2'(X_1')].$$

The circuit diagram and the binary tree representing the formation of the pass implicants are shown in Figs. 3.2 and 3.3 respectively. As seen in Fig. 3.3, four comparisons are needed in the first iteration. The remaining iterations require two and one comparisons, respectively. Thus, a total of only seven comparisons are required to implement the above function. It is obvious that the number of iterations will not exceed three.

The procedure developed above to generate the pass prime implicants can be understood by noting that the minimization procedure follows a straight path on a binary

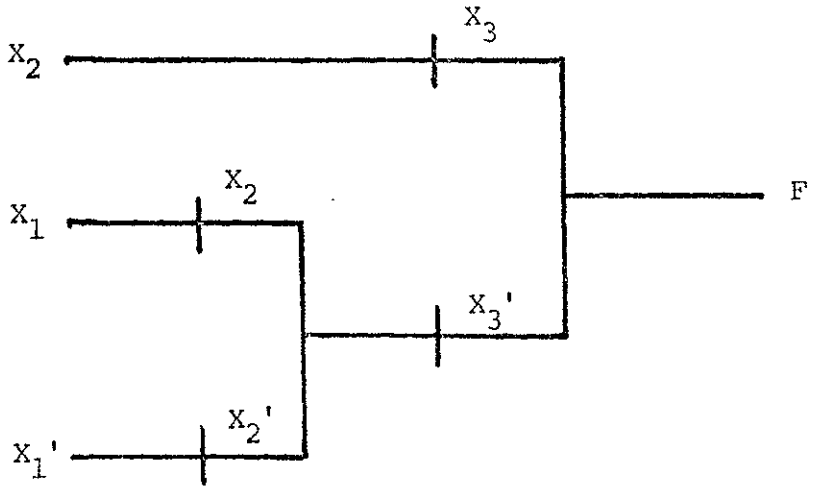


Fig. 3.2. BTS pass Network for Example 3.1.

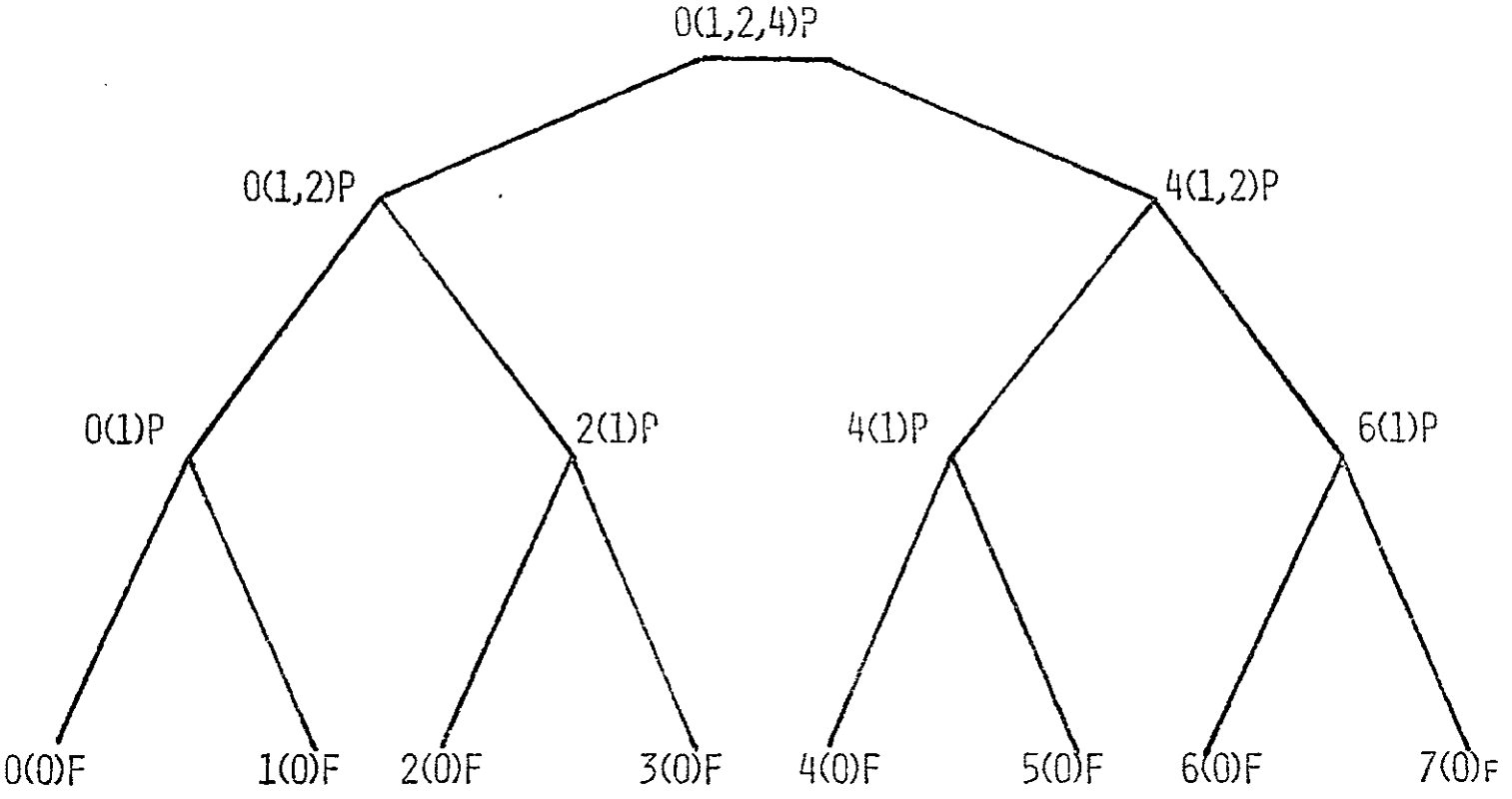


Fig. 3.3. Binary Tree for a 3 Variable Function.

tree as shown in Fig. 3.4. The comparison starts from the bottom leaves of the tree and progresses upward to its root node to arrive at the pass prime implicants. Note that only the B and the D fields of each record are shown in Fig. 3.4.

3.4 Complexity of the Algorithm

There are some advantages to this algorithm which make the algorithm efficient and fast. The number of comparisons used to arrive at the pass prime implicants is almost minimal as the following lemma shows:

Lemma 3.1: The BTS pass network algorithm for an n variable function uses at most $2^n - 1$ comparisons and n iterations.

Proof: Consider again the binary tree shown in Fig. 3.4. The minimization procedure starts from the bottom of the tree. The tree has 2^n leaves. In the first iteration, all the 2^n leaves are compared in pairs for possible combination thus amounting to 2^{n-1} comparisons. In a similar manner in the next iteration at most 2^{n-2} comparisons may be required. The algorithm always terminates after the last iteration where the number of nodes is two. Hence the total number of comparisons needed is: $T = 2^{n-1}+2^{n-2}+\ldots+2^0 = 2^n - 1$.

The number of iterations is also equal to the number of variables because each variable corresponds to one level of the tree or one iteration. QED.

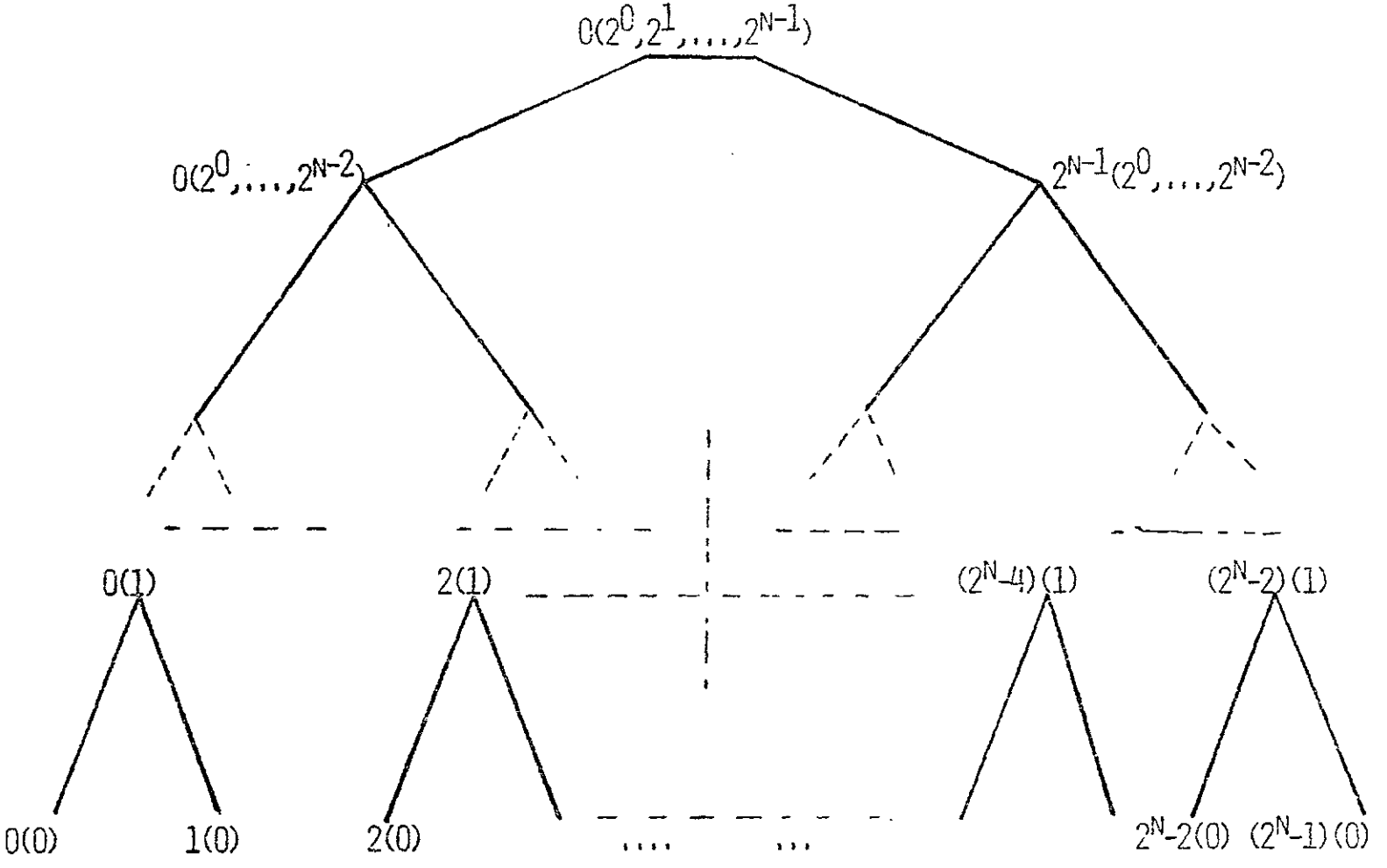


Fig. 3.4. Binary Tree for an n Variable Function.

The memory requirements for the above algorithm is worth mentioning. Implementation of the above algorithm on a computer requires only three arrays, each of size $(3/2)2^n$.

The comparisons of this algorithm with the one given in [1] shows a number of advantages. In [1], the number of comparisons is almost equal to 2^{n**2} and the number of iterations is increasing quadratically with n.

It is however important to note that the above algorithm is not optimal. But this can be justified by the fact that the interconnection topology in today's LSI/VLSI systems is far more important than the number of transistors used to implement a function. The minimum transistor implementation of a function often requires much more surface area for its layout than does an alternative design using more transistors but having simpler interconnection topology [5]. The BTS pass networks have simple interconnection topology as well as minimal transistor count for their implementation.

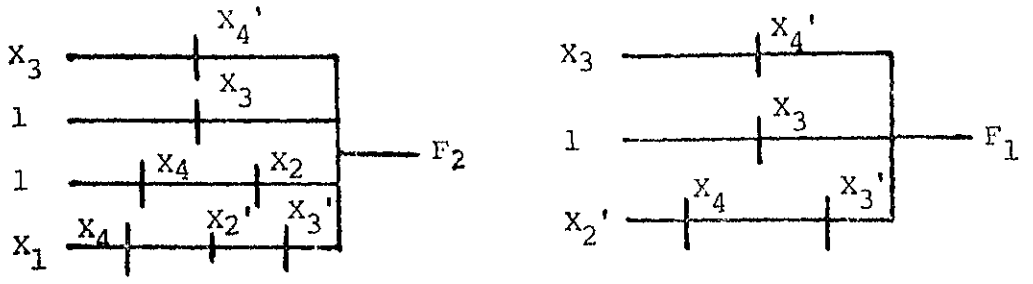
CHAPTER 4

MULTIPLE-OUTPUT PASS NETWORKS

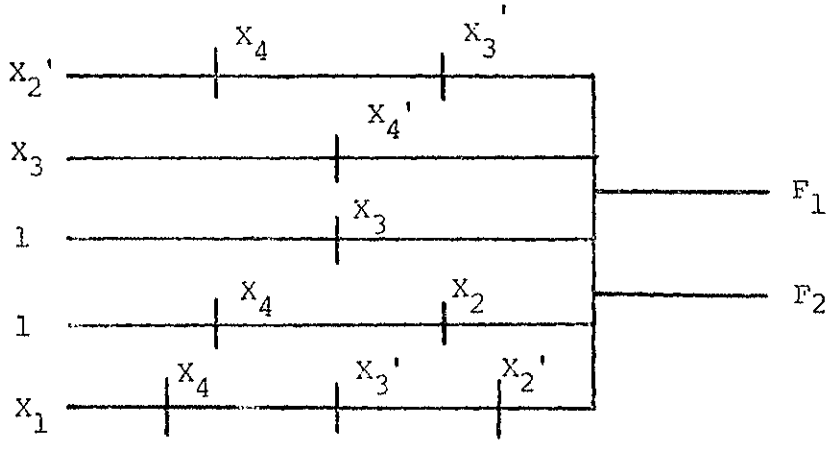
The design of multiple-output networks in pass logic is more involved in comparison to gate logic networks because of the bilateral nature of pass transistors. All transistors involved in the common pass terms cannot be shared since this will involve shorting the outputs of different functions which share this pass term. This is illustrated by considering two functions F_1 and F_2 where $F_1 = X_4'(X_3)+X_3(1)+X_4X_3'(X_2')$, and $F_2 = X_4'(X_3) + X_3(1) +X_4X_2(1)+ X_4X_3'X_2'(X_1)$ shown in Fig. 4.1. If the common portion $[X_4'(X_3) + X_3(1)]$ is shared to form a multiple-output pass network as shown in Fig. 4.1(b), the two functions get shorted at all times. Hence additional conditions must be satisfied in sharing among different pass networks.

4.1 Sharing in Pass Networks

When portions of a pass network is shared among different functions it is necessary to ensure that the functions are isolated during times when the shared part is not passed to their outputs. The following theorem gives



(a)



(b)

Fig. 4.1. Improper Sharing Between Two Functions.

the necessary and sufficient conditions for sharing a pass network among different pass functions.

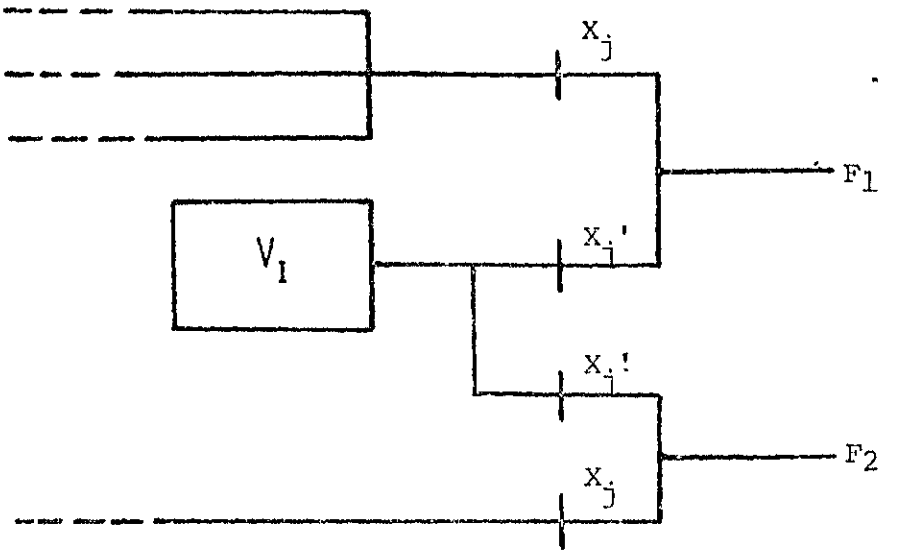
Theorem 4.1: A pass function,

$F = P_1(V_1)+P_2(V_2)+\ldots+P_i(V_i)+\ldots+P_n(V_n)$, can share V_i with other functions iff for some variable X_j^* (X_j^* is either X_j or X_j') in P_i,

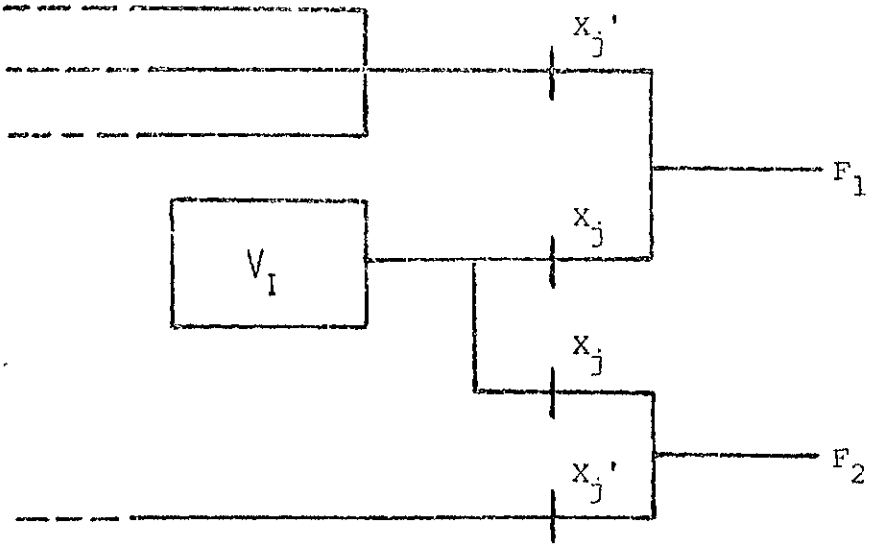
1. $X_j^*(V_i)$ appears in all the shared functions, and
2. $(X_j^*)'$ occurs in P_j for all $j = i$.

Proof: Consider part of a pass function as shown in Fig. 4.2. The pass networks shown in Fig. 4.2 are the only two cases where X_j is common to all the terms satisfying the complementary property. In Fig. 4.2(a) $X_j'(V_i)$ is common to both F_1 and F_2; When $X_j'=1$, V_i is passed to both the functions F_1 and F_2. All the other paths to the output in both the functions are open due to the pass transistors controlled by $X_j=0$; When $X_j=1$, V_i is disconnected from both F_1 and F_2 and the two functions are separated from each other. This will ensure that there will be no conflict in either network. The network shown in Fig. 4.2(b) functions in an analogous manner. Thus both functions can share V_i.

If V_i is shared between two functions, this implies that V_i must be passed to both the functions when the corresponding p-term is 1 and the two functions must be separated from each other when the p-term is 0. The former condition requires that V_i be connected through a pass transistor, controlled by a variable X_i^*, to both the



(a) For $X_j = 0$, $F_1 = F_2 = V_i$



(b) For $X_j = 1$, $F_1 = F_2 = V_i$

Fig. 4.2. A Shared Multiple-Output Pass Network.

functions, or, in other words, $X_i^*(V_i)$ be common to both the functions. The latter condition requires that when V_i is passed to the output, none of the other pass variables be passed to the output simultaneously, thus requiring that pass term V_i and all the other pass terms V_j for $j = i$ be switched by disjoint product terms P_i and P_j. QED.

The following example can be used to illustrate the above algorithm.

Example 4.1: Consider the two functions F_1 and F_2 given by:

$F_1(X_4X_3X_2X_1) = m(1,5,6,7,8,12,13,14,15)$, and

$F_2(X_4X_3X_2X_1) = m(1,2,3,5,6,7,8,10,11,12,13,14,15)$.

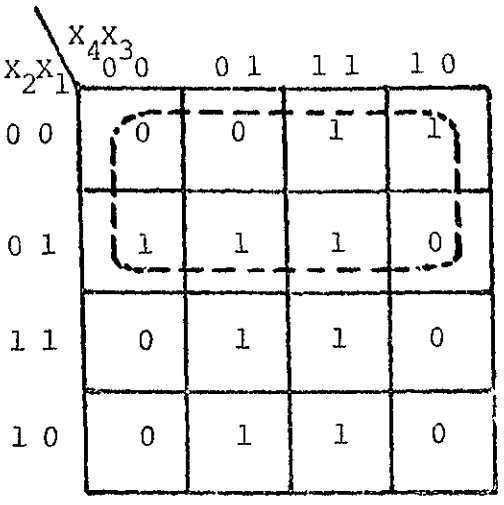
The corresponding pass functions are:

$F_1 = X_2(X_3)+X_2'[\ X_4X_1(X_3)+X_4'(X_1)+X_1'(X_4)\ ]$, and

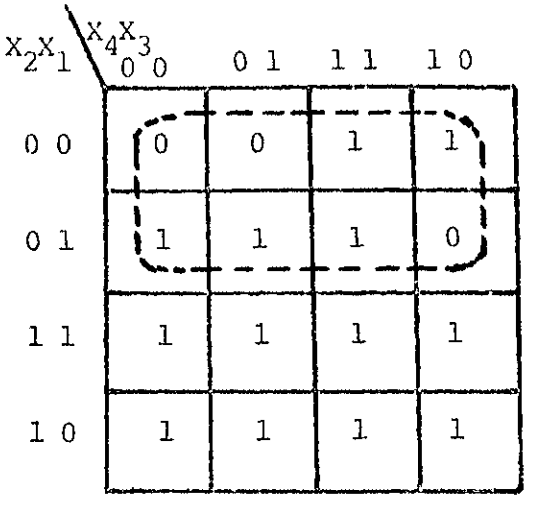
$F_2 = X_2(1)+X_2'[\ X_4X_1(X_3)+X_4'(X_1)+X_1'(X_4)\ ]$.

Their circuit diagrams and K-maps are shown in Fig. 4.3. The shared part in the two functions corresponds to the upper eight cells as marked in the K-maps. The shared part $X_1'(X_4)+X_4'(X_1)+X_4X_1(X_3)$ is connected through X_2' to both the functions. All the other paths in both F_1 and F_2 go through X_2 thus satisfying the conditions of Theorem 1. As seen from Fig. 4.3(c) the outputs F_1 and F_2 satisfy the K-maps under all input conditions.

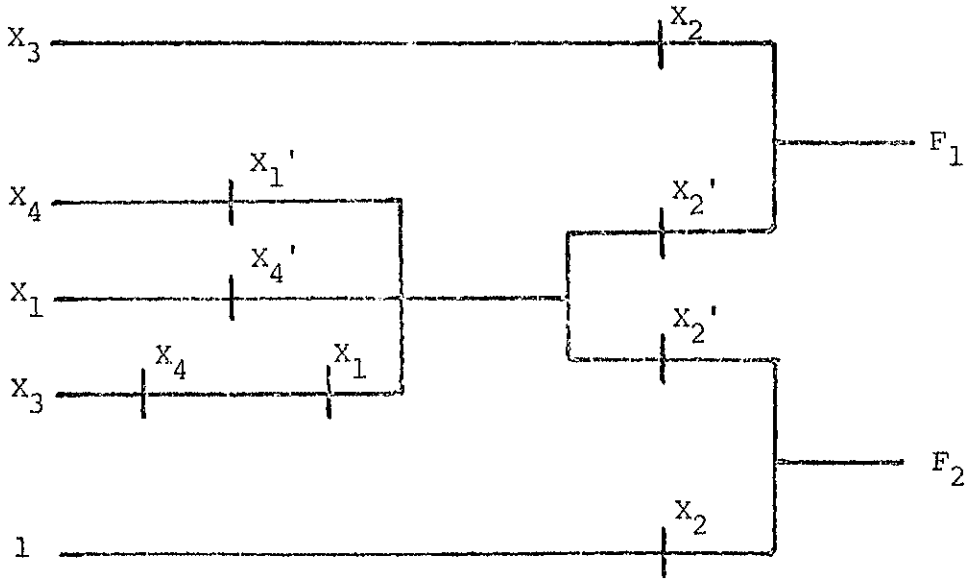
A special case of the above theorem involves sharing part of a pass network between two functions. In this case, some of the conditions given above can be relaxed.



(a) K-Map for F_1



(b) K-Map for F_2



(c)

Fig. 4.3. Multiple-Output Pass Network of Example 4.1.

Theorem 4.2: A pass function, $F_1 = P_1(V_1)+P_2(V_2)+..+P_i(V_i)+...+P_n(V_n)$, can share V_i with function F_2 if for some variable X_j in P_i, $X_j(V_i)$ appears in function $F_1(F_2)$ and $X_j'(V_i)$ appears in $F_2(F_1)$.

Proof: Consider two pass functions F_1 and F_2 as shown in Fig. 4.4. $X_j(V_i)$ appears in F_1 and $X_j'(V_i)$ appears in F_2. Even though V_i is shared by both F_1 and F_2, it is not passed to both of them simultaneously. The connections between F_1 and F_2 is only through the series connection of the transistors controlled by X_j and X_j'. Therefore no activated part of one function can access the nodes of the other function. When V_i is not passed to either output function, F_1 is still isolated from F_2. Thus V_i can be shared between the two functions. QED.

Example 4.2: Consider two functions F_1 and F_2 given by:

$F_1(X_4X_3X_2X_1) = m(0,1,3,8,9,12,13)$, and

$F_2(X_4X_3X_2X_1) = m(8,9,11)$. The resulting pass functions are:

$F_1 = X_4(X_2')+X_4'[X_3(0)+X_2'(X_3')+X_3'X_2(X_1)]$, and

$F_2 = X_4'(0)+X_4[X_3'X_2(X_1)+X_2'(X_3')+X_3(0)]$.

The circuit implementation and K-maps for the above functions are shown in Fig. 4.5. The shared part of the two functions encircled in the K-maps correspond to two different locations on the maps, namely X_4 and X_4'. Here again X_4 is used to prevent a conflict as the result of a short circuit.

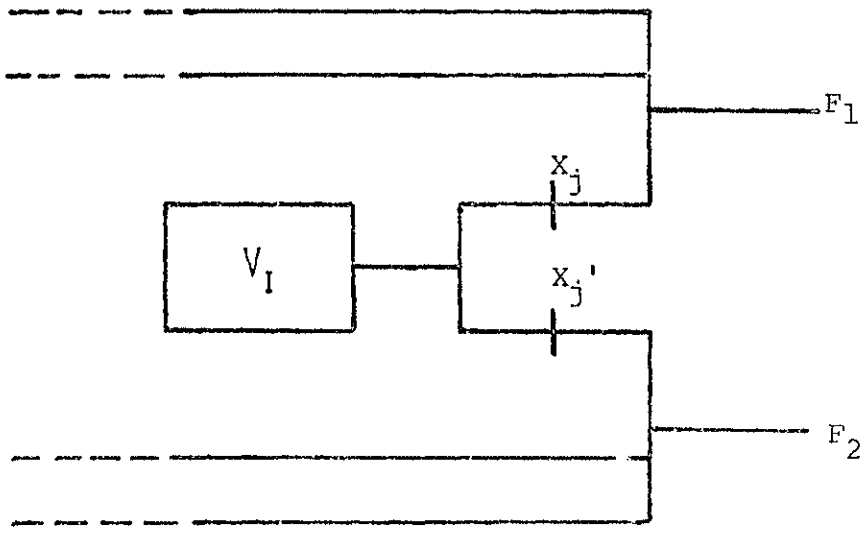
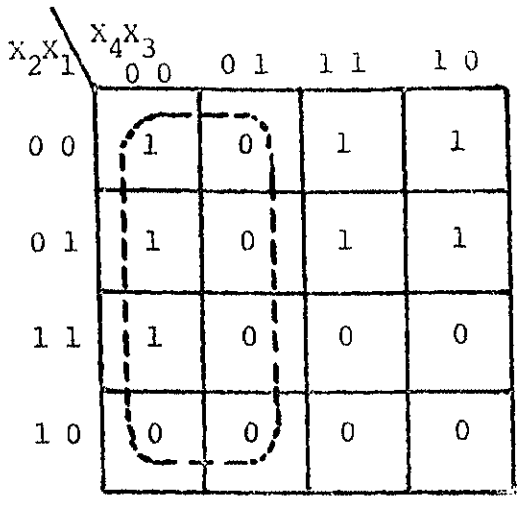
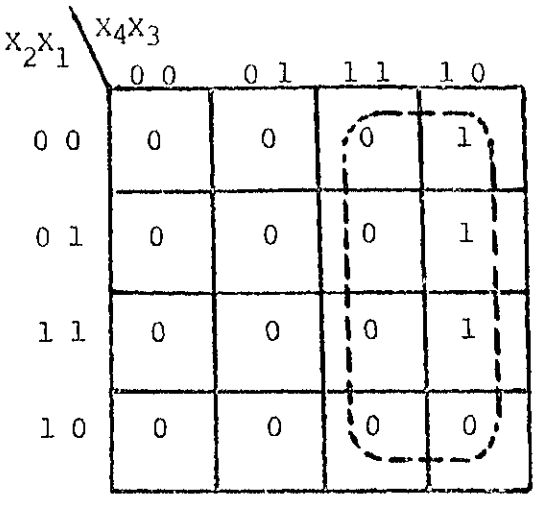


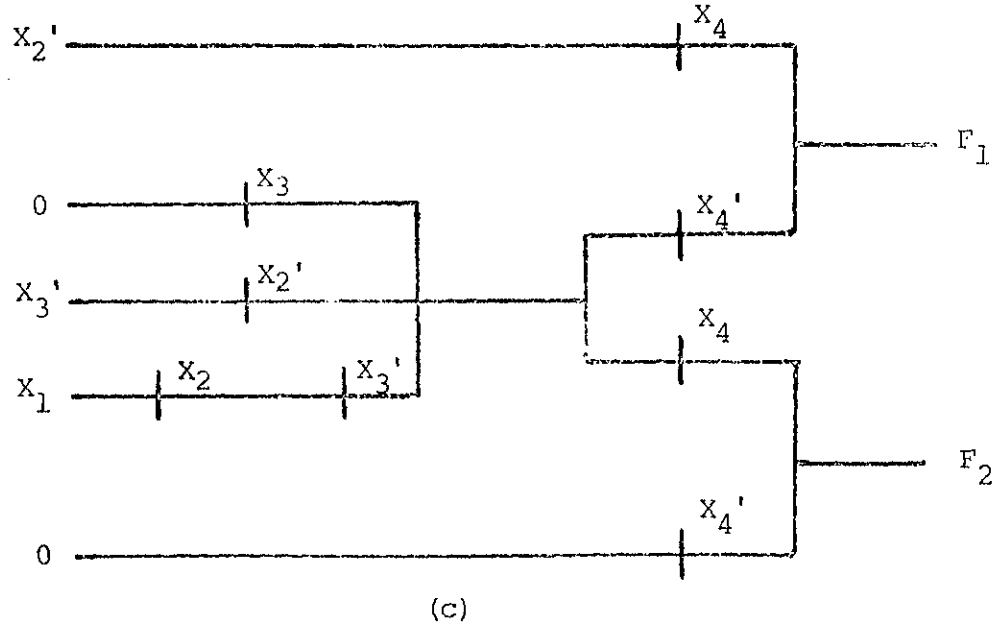
Fig. 4.4. Sharing V_i Between Two Functions F_1 & F_2.



(a) K-Map for F_1



(b) K-Map for F_2



(c)

Fig. 4.5. Multiple-Output Pass Network of F_1 & F_2.

Condition 2 of Theorem 1 is always satisfied for BTS pass networks. Thus BTS pass networks are well suited for realizing multiple-output pass functions.

4.2 Multiple-Output BTS Pass Networks

The property of BTS pass networks allows us to relax many of the conditions needed for sharing part of a network among several pass networks.

Definition 4.1: A complementary sum (CS) S_i is defined as $S_i = X_i(V_j)+X_i'(V_k)$ where X_i is any control variable and V_j and V_k are either pass variables or complementary sums by themselves.

By the definition of CS, a BTS pass function can always be expressed as a CS. To illustrate this, consider a BTS pass function $F = X_4[\ X_3(X_1')+X_3(0)\ ]+X_4'[\ X_2(1)+X_2'(X_1)\ ]$. The CSs in this function are : $V_1 = X_3(X_1')+X_3'(0)$ and $V_2 = X_2(1)+X_2'(X_1)$. Therefore, the BTS pass function F can be written as: $F = X_4(V_1)+X_4'(V_2)$ which is in CS form.

Definition 4.2: If a complementary sum S_i in a BTS pass network is common to several BTS pass networks, Then S_i is called a shared complementary sum (SCS).

The following theorem gives the necessary and sufficient conditions for sharing a CS among several BTS pass functions.

Theorem 4.3: A BTS pass function F_1 can share part of its network S_i with other functions iff S_i is common to all the functions.

Proof: The complementary requirements of the BTS pass function imply that all the conditions of Theorem 1 and 2 are satisfied and therefore the proof of this theorem follows directly from Theorem 1 and 2.

Example 4.3: Consider implementation of the following functions as multiple-output BTS pass function.

$F_1(X_4X_3X_2X_1) = m(0,1,2,3,4,5,6,7,9,10,11,14,15)$

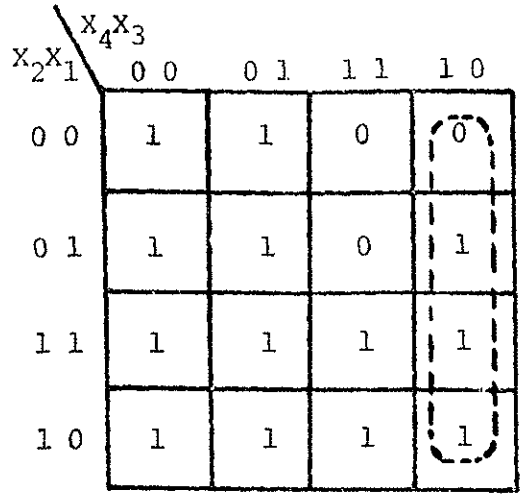
$F_2(X_4X_3X_2X_1) = m(1,2,3,5,6,7,11,15)$

The minimization of these two functions in BTS form results in $F_1 = X_4'(1)+X_4[\ X_3(X_2)+X_3'[\ X_2(1)+X_2'(X_1)\ ]]$, and $F_2 = X_4'[X_2(1)\ +X_2'(X_1)]+X_4[X_2(X_1)+X_2'(0)]$. The K-maps and the multiple-output pass network for the functions F_1 and F_2 are shown in Fig. 4.6. The complementary sum $S_i = X_2(1)+X_2'(X_1)$ is shared between the two functions. When $X_4 = 0$, $F_2 = S_i$. Similarly, when $\langle X_4X_3\rangle = \langle 10\rangle$, $F_1 = S_i$. The structure of the network prevents any interaction between the two functions F_1 and F_2.

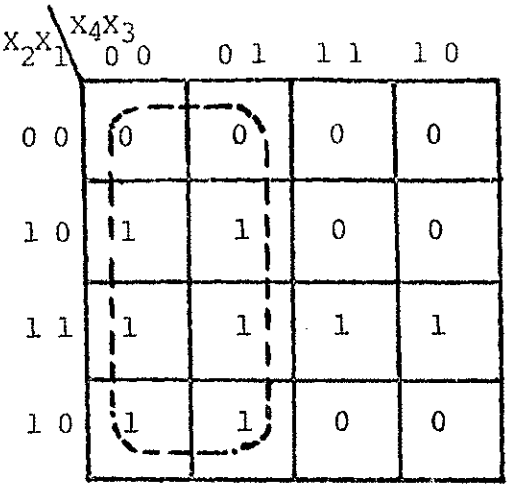
4.2.1 Design of Multiple-Output BTS Pass Networks

The BTS algorithm given in Chapter 3 is based on a single- output function. The modification to include multiple-output BTS pass networks is illustrated as follows.

Consider the functions $F_1 = X_1(V_i) + X_1'[\ X_2(V_d)+X_2'[X_3(V_n)+X_3'(V_m)\ ]\ ]$ and $F_2 = X_2(V_k) + X_2'[\ X_3(V_n)+X_3'(V_m)\ ]$. First the two functions are simplified separately using the earlier algorithm. The CS $S_i = X_3(V_n)+X_3'(V_m)$ is to both functions and can be shared. In order to find the SCS



(a) K-Map for F_1



(b) K-Map for F_2

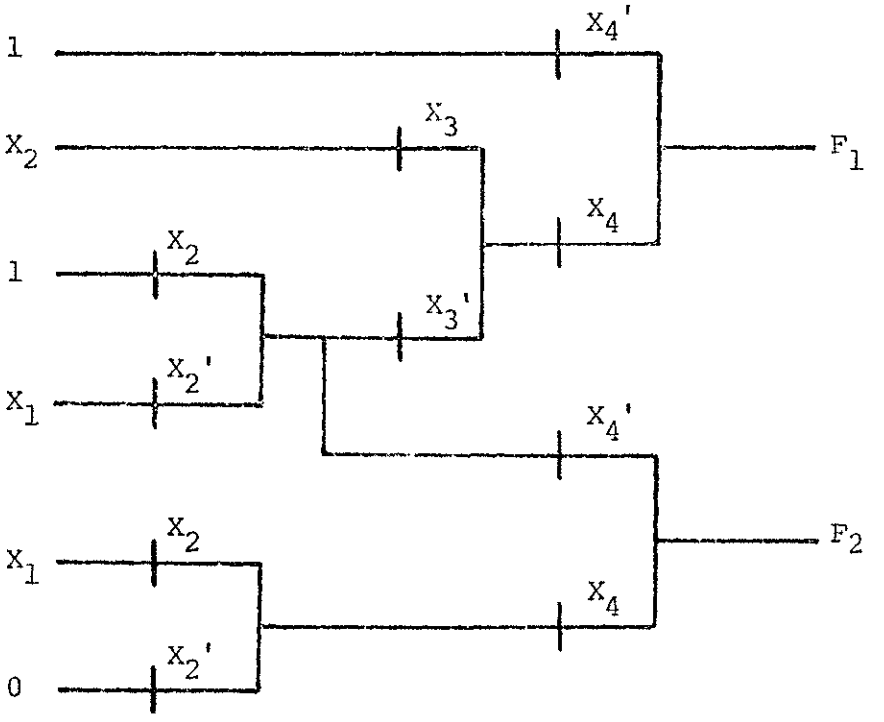


Fig. 4.6. A Multiple-Output BTS Pass Network.

of several given functions, the comparisons must start with largest CS in one of the functions. Then determine if this CS is common to any one or more of the other functions. If it is common to any other function, then this CS is an SCS, otherwise choose a smaller CS, if it exists, in the same function and do the comparison again. Repeat the above procedure until all the CSs in the function are compared. Continue the same procedure to the rest of the given functions until all the SCSs of multiple-output BTS pass networks are found.

CHAPTER 5

PROGRAMMABLE LOGIC ARRAYS (PLA)

A programmable logic array can, in general, be defined as a multiple-output network. A PLA is used to map irregular combinational functions onto regular structures which allows significant changes in the functions without requiring major changes of either the design or its layout. Conventional design of a PLA uses the AND-OR plane approach. For LSI/VLSI designs NOR forms are preferred. A new approach for the design of a PLA using pass logic is presented here.

5.1 Stick Diagram Representation

In this section some notations and the design rules used in current VLSI technology [5] are presented. A stick diagram representation is used here to show the structure of an NMOS PLA. In stick notation each silicon layer in an IC chip is represented by a colored line, thus each layer will be referred to as a line. The diffusion, polysilicon, and metal layers are assigned green, red, and blue colors respectively. The lines of different colors crossing each other has no effect wheras lines of the same color imply a connection. A special case is when polysilicon crosses

over diffusion which produces an enhancement mode MOS transistor. A connection between two different layers is indicated by a black dot at the point of intersection. Stick notations for some commonly used devices are shown in Fig. 5.1.

5.2 Gate logic PLA:

Figure 5.2 illustrates the overall structure of a PLA [5]. The two registers added to the structure makes it easy to modify the PLA to a finite state machine.

The inputs go to the AND plane. The different implicants generated by the AND gates in the AND plane are passed to the OR plane and the outputs are taken out of this OR plane. In LSI/VLSI designs a NOR-NOR PLA is preferred over the AND-OR PLA [5]. The circuit diagram of an example PLA and its stick diagram are given in Fig. 5.3 and 5.4. The different outputs from the PLA are:

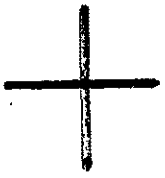
$$F_1 = X_4'X_2' + X_4X_3X_1' + X_4X_3'X_1$$

$$F_2 = X_4'X_2' + X_4X_3X_2$$

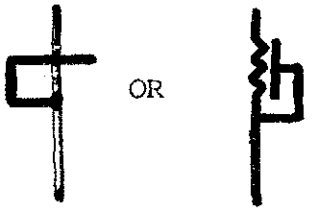
$$F_3 = X_4X_3X_2 + X_4X_3'X_2'.$$

5.3 Pass Logic PLA

The design of a pass logic PLA is different from the above AND-OR PLA because of the nature of pass networks which consist of tri-state devices, MOS pass transistors. First, both the 0's and 1's of a function must be used in implementing a pass network which in turn will enlarge the



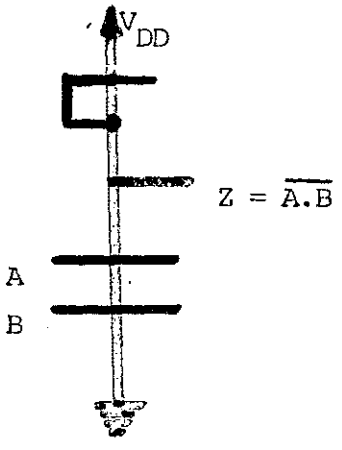
(a) An NMOS Enhancement Mode Transistor



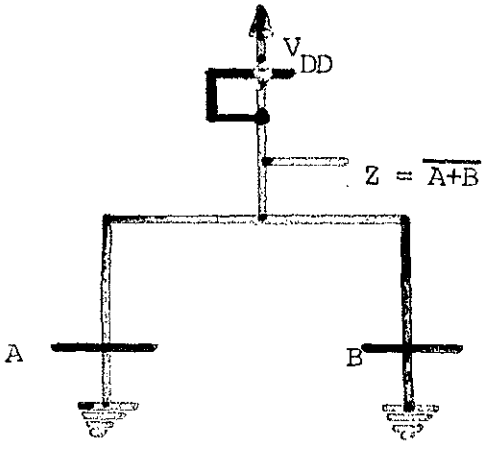
(b) An NMOS Depletion Mode Transistor

(c) A Connection in Diffusion

(d) A Connection Between Diffusion and Metal



(e) A 2-Input NAND Gate



(f) A 2-Input NOR Gate

Fig. 5.1. Stick Diagrams.

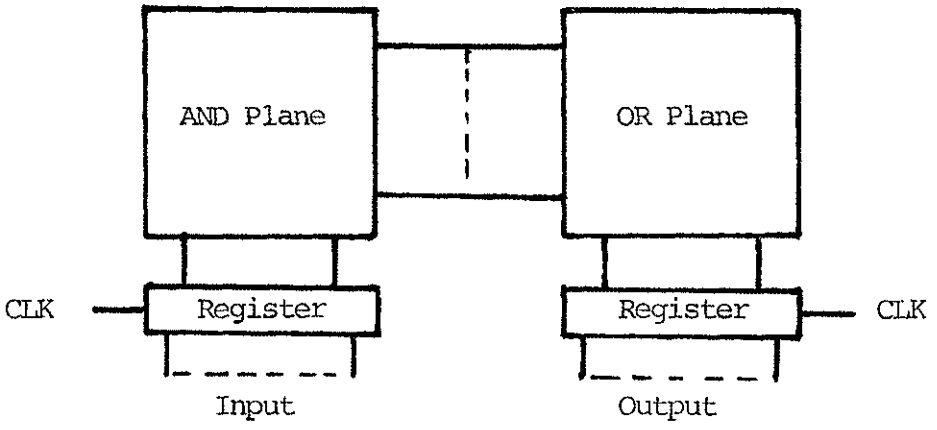


Fig. 5.2. Overall Structure of PLA.

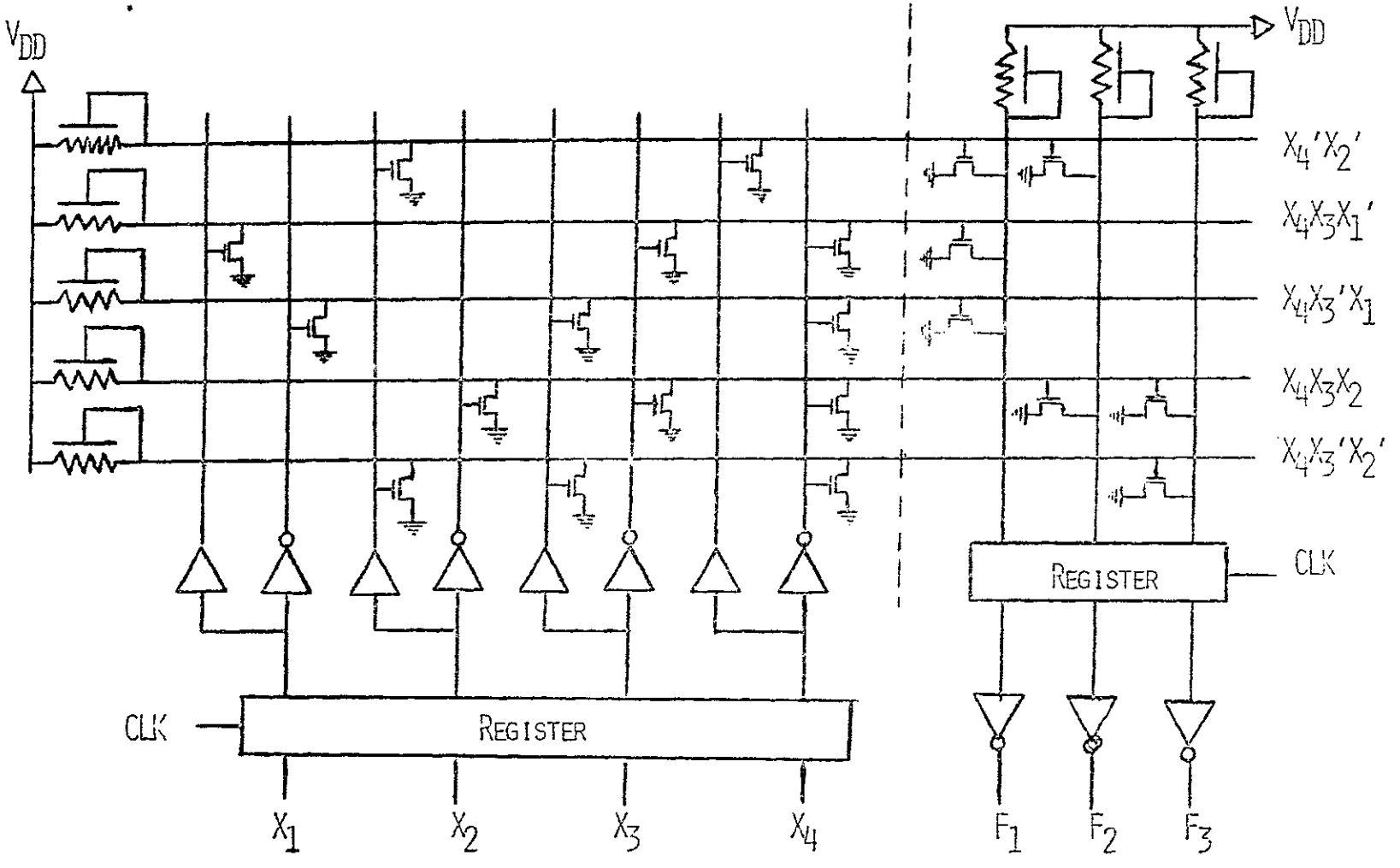


Fig. 5.3. Circuit Diagram of a Gate Logic PLA.

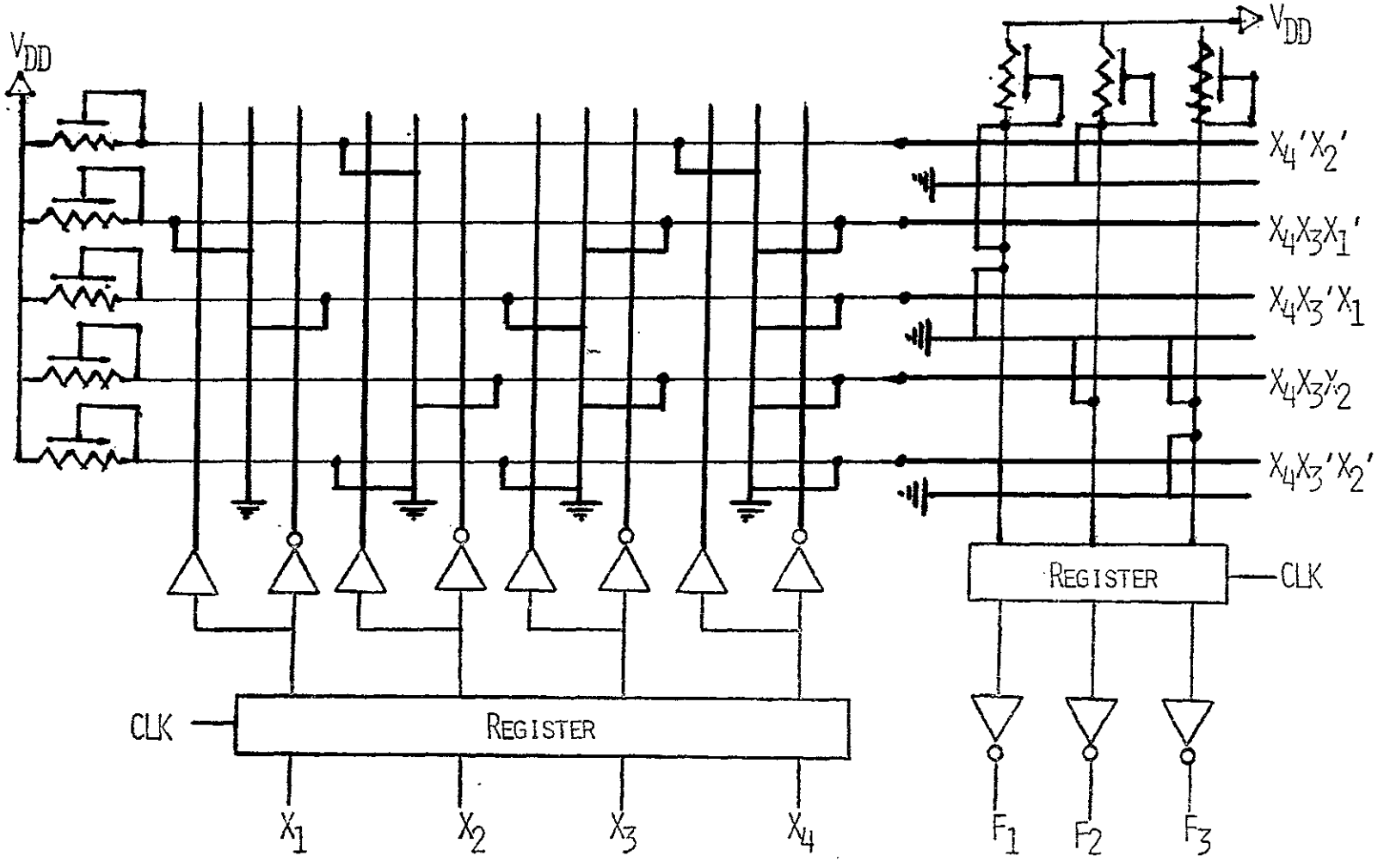


Fig. 5.4. Stick Diagram of a Gate Logic PLA.

size of the PLA layout. Second, as mentioned in previous sections, the sharing of a pass implicant by several functions will short outputs of the functions. The first problem may, in some cases, be solved if the functions are minimized before PLA implementation. The second problem can be solved by devising a circuit to detect a high impedance state at the outputs of the transistors. Such a circuit can be used to separate the outputs of the functions which share common pass implicants.

A pass network PLA layout is similar to the conventional PLA. It consists of an AND plane where the different pass implicants are formed. To generate the final output, the control and the data lines in the AND plane are routed perpendicular to each other as in the conventional PLA. The implementation of an n-variable multiple-output function in this manner requires 2^n vertical lines if the functions are not minimized. Note that each horizontal line (pass implicant) may have three states 0, 1, or high impedance. Therefore, in the design of a pass logic PLA care must be taken not to short the outputs of different functions which share a pass implicant. The outputs of the functions which share a pass implicant must be isolated from one another at all times except when the shared pass implicant is enabled to the outputs. A CMOS (combination of NMOS and PMOS) transistor can be used to isolate the outputs at proper times. The CMOS transistor determines whether the outputs must be

shorted or not depending on the state of the shared pass implicant line. If the shared pass implicant is in a high impedance state, then the corresponding outputs are isolated; otherwise, outputs are shorted. The CMOS transistors must have very high threshold voltage so that as soon as the voltage input is decreased (high impedance), the transistors are cut-off. The following example illustrates the proposed pass logic PLA.

Example 5.2: Consider the implementation of the following minimized functions as a pass logic PLA.

$F_1 = X_4'(X_2') + X_4X_3(X_1') + X_4X_3'(X_1)$,

$F_2 = X_4X_3(X_2) + X_4X_3'(0)$, and

$F_3 = X_4'(0) + X_4X_3(X_2) + X_4X_3'(X_2')$

The stick diagram corresponding to the above functions are shown in Fig. 5.5.

Note that if the functions are not minimized before implementation as a pass logic PLA, the size of the layout will increase drastically compared to a gate logic PLA. The speed of the proposed PLA also depends on the number of transistors in each horizontal line. If the number of transistors increases by four, a buffer must be put between each succeeding four transistors to avoid long delay. Therefore, this PLA is good for implementation of small functions. Thus a gate logic PLA for implementation of large functions is faster and more economical than a pass logic PLA.

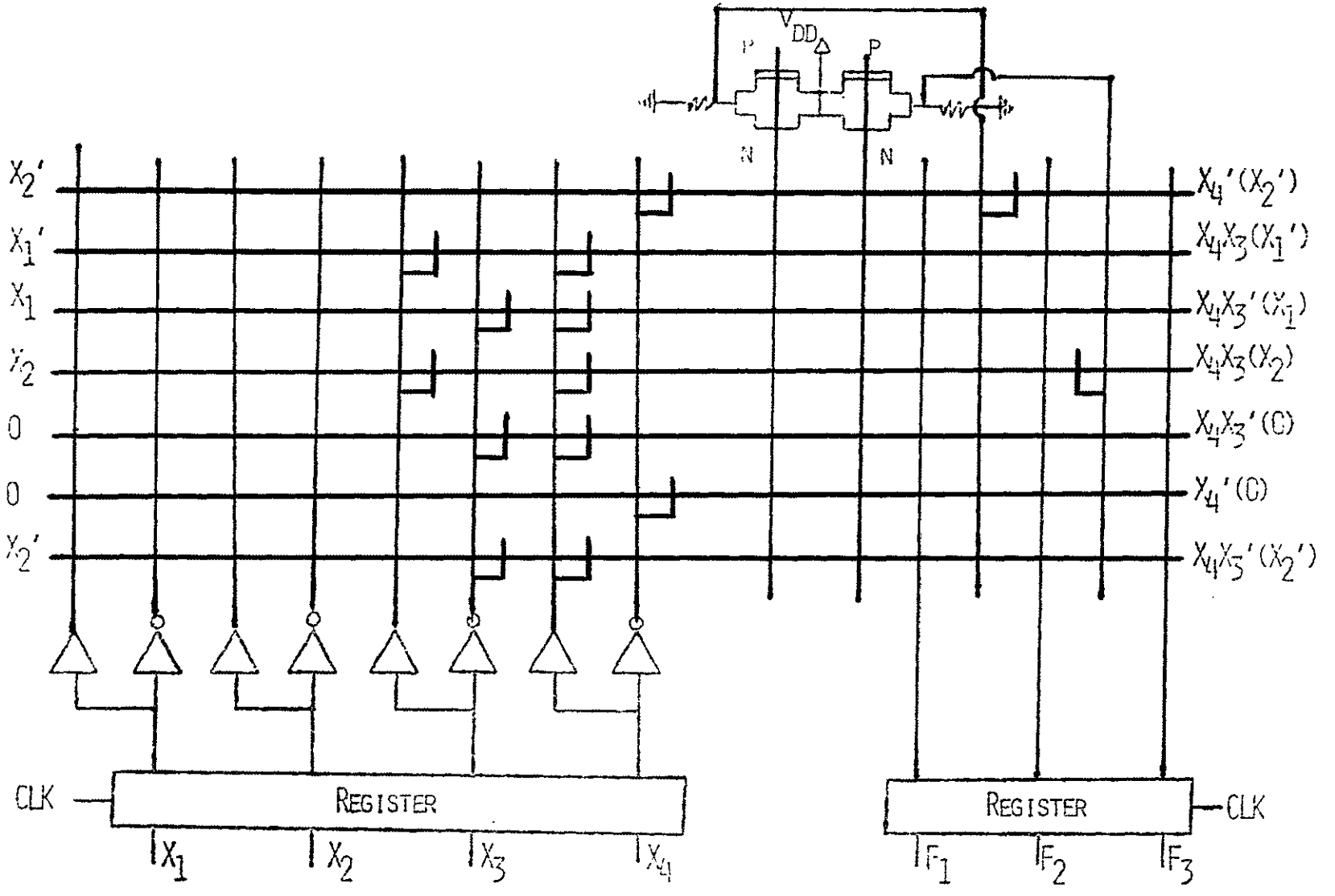


Fig. 5.5. Stick Diagram of a Mask Logic PLA.

CHAPTER 6

FAULT DETECTION IN MULTIPLE-OUTPUT PASS NETWORKS

Fault modeling in pass networks considered in [2,3,4,6,7] shows that conventional stuck-at (s-at) fault models alone are not sufficient for the modeling of pass network faults because pass transistors are tri-state devices. Pass networks exhibit a new kind of fault called stuck-open (s-op) [3,4,6,7]. In addition, a node may be connected to both V_{DD} and GND at the same time called stuck-on (s-on), under a faulty situation. The test sets derived in [2,4] are based only on single-output pass networks. But the results derived in this are true for both single and multiple-output pass networks. All faults in a BTS pass network are shown to be equivalent to stuck-at faults on pass variables, and s-on and s-op faults occurring in the pass transistors [2,4].

The detection of faults in multiple-output pass networks is similar to that in single-output circuits. Thus, the techniques used to test single-output pass networks can also be used to detect faults in multiple-output pass networks.

6.1 Fault Detection

The fault detection procedures developed in this section are based on disjoint pass networks [4]. Pass implicants in these networks are disjoint and they also possess good fault detection properties.

For test purposes those portions of a multiple-output pass network excluding the shared part V_i can be treated as single-output pass networks. This is illustrated by considering the pass network shown in Fig. 6.1. The test vectors can be applied to any path in the upper two branches of the function F_1 and its output can be observed at F_1. Similarly, test vectors can be applied to any path in the lower two branches of the function F_2 and the output can be observed at F_2.

Lemma 6.1: A shared network V_i of a multiple-output pass function can be tested by applying proper test vectors which enable V_i to the output of one of the functions and by observing the changes occurring at the output of that function.

Proof: It is obvious, as shown in Fig. 6.1, that a fault in V_i, can be detected by either setting X_j equal to 1 and by observing changes at the output of the function F_2 or by setting X_j equal to 0 and by observing the changes at the output of the function F_1. These tests can always be done because only one path is enabled at a time for a fault-free pass network with disjoint pass implicants. QED.

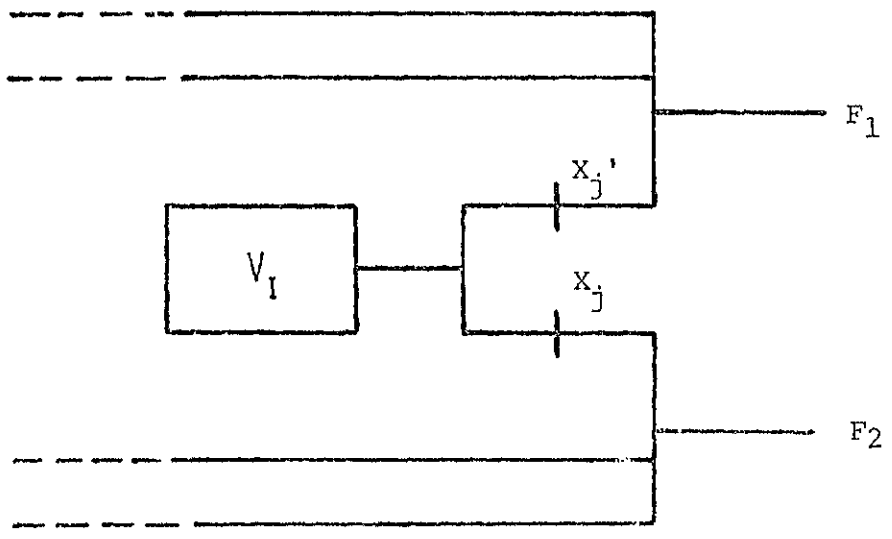


Fig. 6.1. A Two Output Network sharing V_i.

Note that enabling V_i to the output of one function may also result in enabling V_i to the output of other functions. Whenever this occurs, a fault in several functions may be tested simultaneously by applying a single set of test vectors.

6.1.1 Pass Variable Faults

The faults occurring in a pass variable are classified into either s-at or s-op. A s-at fault at a node behaves as if the node is connected at a fixed logic state determined by the fault. A s-op fault in a network branch behaves like a discontinuity in the branch, thus making the output float (high impedance state). The behavior of a s-op fault is equivalent to a s-op in the connecting transistor.

Lemma 6.2: A s-at fault in the pass variable V_i in S_i of a multiple-output BTS pass network is detectable by sensitizing V_i to any one of the outputs.

Proof: If V_i is sensitized to one of the outputs F_i, then $F_i = V_i$. If V_i is constant, then the output F_i determines the existance of a s-at fault on V_i. If V_i is a variable then s-at 1/0 can be detected by feeding V_i with a 0/1 value.

Theorem 6.1: All pass variable s-at faults in a BTS pass network are detectable.

Proof: In a multiple-output BTS pass network one and only one path is enabled to a single output at any time. Hence by Lemma 6.2 all s-at faults are detectable. QED.

The following example illustrates the above theorem.

Example 6.1: Consider the multiple-output BTS pass network shown in Fig. 6.2. Two sets of test vectors for testing the s-at faults in the pass variables of the shared complementary sum $S_i = X_2(1) + X_2'(X_1)$ are shown in Table 6.1- one by observing the output F_1, and the other by observing the output F_2. Any one of these two sets of test vectors is sufficient to test all the pass variable faults in S_i. The test vectors for testing all other pass variables are shown in Table 6.2.

6.1.2 Pass Transistor Faults

The faults which affect the pass transistors are modeled as s-op and s-on [2,4]. A s-on fault between two nodes in a network behaves like a short circuit between them. This may cause logical inconsistency if both a '0' and a '1' are passed simultaneously through these two nodes. If a s-op fault exists along an enabled path, the output will float and enter into a high impedance state. Due to the capacitive loading in the circuit, the output will remain in the previous state at least for a short period of time depending on the time constants involved in the circuit. To detect this fault, the output must be forced to make a transition between 0 and 1.

Theorem 6.2: All s-op faults in a BTS pass network are detectable.

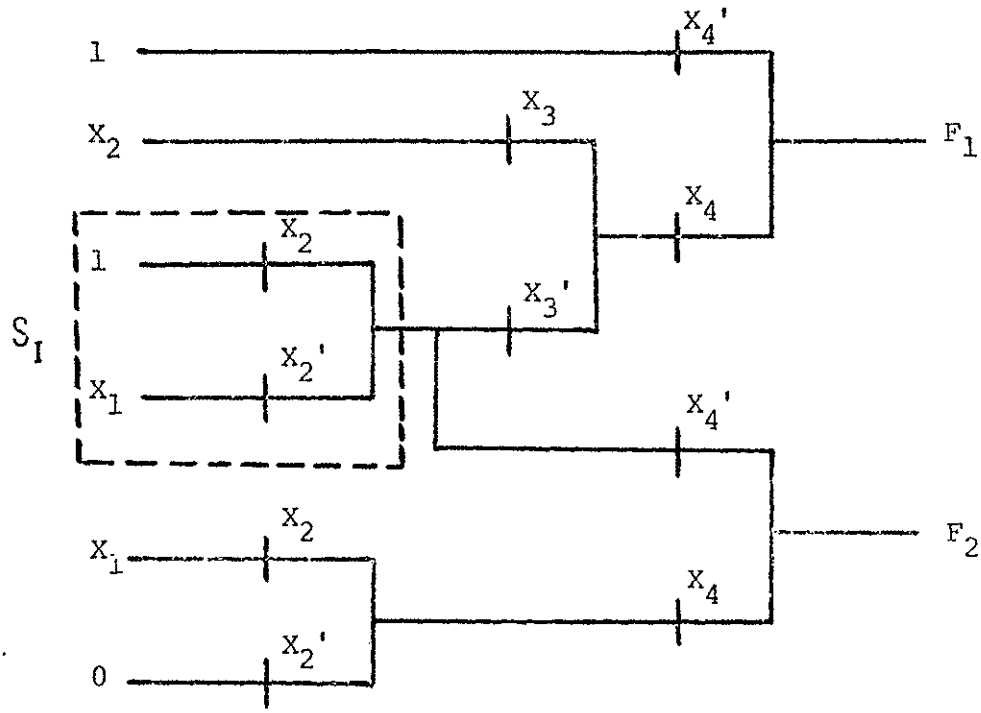


Fig. 6.2. S-AT Pass Variable Faults in a Multiple-Output BTS Pass Network.

S-at Fault	F_1				F_2			
	X_4	X_3	X_2	X_1	X_4	X_3	X_2	X_1
1 @ 0	1	0	1	-	0	-	1	-
X_1 @ 0	1	0	0	1	1	0	0	0
X_1 @ 0	1	0	0	0	1	0	0	0

Table 6.1. Pass Variable S-at Fault Test Vectors for S_i of Fig. 6.2.

S-at Fault	F_1				F_2			
	X_4	X_3	X_2	X_1	X_4	X_3	X_2	X_1
X_1 @ 0					1	-	1	1
X_1 @ 1					1	-	1	0
0 @ 1					1	-	0	-
X_2 @ 0	1	1	1	-				
X_2 @ 1	1	1	0	-				
1 @ 0	0	-	-	-				

Table 6.2. Pass Variable S-at Fault Test Vectors in Fig. 6.2 Excluding S_i.

Proof: A s-op fault is tested by feeding two sets of inputs, the first one initializes the output to 1/0 through a fault-free path and the second one passes a 0/1 through the faulty path. The basic property, that all the pass variables in a pass network are not identical, allows us to find two inputs satisfying the above condition and hence the theorem. QED.

Note that the above theorem is also valid for s-op faults in pass variables and internal nodes.

The following example illustrates the above theorem.

Example 6.2: Consider once again the multiple-output BTS pass network shown in Fig. 6.2. For identifying the different transistors for test purposes, additional subscripts are added to all the control variables. The modified diagram is shown in Fig. 6.3.

The s-op fault in X_{41}' can be detected in the following manner: Apply the input vector $\langle X_{41}X_{31}X_{21}X_1 \rangle = \langle 1\ 1\ 0\ - \rangle$. The output F_1 is now set to logic '0'. If X_{41} is now changed to '0', with no fault, the output F_1 must change to '1'. Otherwise the output will stay at the previous logic value of '0'. The initialization can also be done by applying the input vector <1 0 0 1>.

The test vectors for all the s-op faults in Fig. 6.3 are given in Table 6.3.

The number of transistors in a long chain is usually limited to four, unless it is separated by buffer stages to avoid excessive delay. In such networks, if a logic '1' is

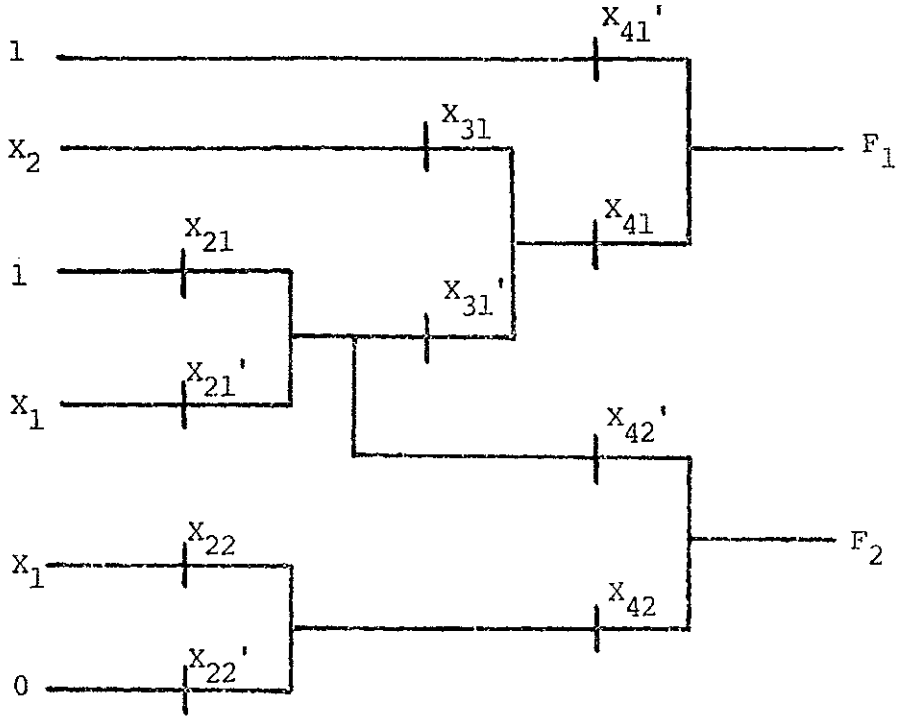


Fig. 6.3. Transistor Faults in Multiple-Output BTS Pass Network.

S-op	1st Test Vector				2nd Test Vector				F_1	F_2
Fault	X_4	X_3	X_2	X_1	X_4	X_3	X_2	X_1		
X_{41}'	1	1	0	-	0	-	-	-	0/1	-
X_{31}	1	0	1	-	1	1	0	-	1/0	-
X_{41}	0	-	-	-	1	1	0	-	1/0	-
X_{31}'	1	1	0	-	1	0	1	-	0/1	-
X_{21}	1	1	0	-	1	0	1	-	0/1	-
X_{21}'	1	-	0	-	0	-	0	1	-	0/1
X_{42}'	0	0	-	0	0	1	-	-	-	0/1
X_{42}	1	-	1	1	1	-	0	-	-	1/0
X_{22}	1	-	0	-	1	-	1	1	-	0/1
X_{22}'	1	-	1	1	1	-	0	-	-	1/0

Table 6.3. S-op Fault Test Vectors for the Network in Fig. 6.3.

fed through a large depletion mode transistor, it can more or less guarantee the dominance of a '0' over a '1'. The following theorem assumes this dominance property.

Lemma 6.3: If one or more pass variables in a BTS pass network are constants, then at least one of the s-on faults is not detectable.

Proof: Consider a general BTS pass network shown in Fig. 6.4. To detect a s-on fault on X_i, we must pass both a '0' and a '1' simultaneously through two different paths such that the '0' is passed through X_i. If X_i is made '0', then the output must be logic 1 if there is no fault, and logic 0 if there is a s-on fault. This will be true only if a '0' dominates a '1'. On the other hand, if a '1' is passed through X_i and a '0' is passed through a second path to the output, then the output will always be '0' independent of the fault in X_i and hence this fault cannot be detected. Similarly, a s-on fault in X_j' cannot be detected since a '0' ia passed to the output through the second path controlled by X_j under test conditions. QED.

Theorem 6.3: All s-on faults in a BTS pass network are detectable if none of the pass variables are constants.

Proof: Since none of the pass variables are constants, their logic values can be chosen arbitrarily. This will allow the logic assignment of pass variables such that a '0' is passed through the faulty path and a '1' through a fault free path to the output simultaneously, thus detecting the fault, All s-on faults can be detected in a similar manner.

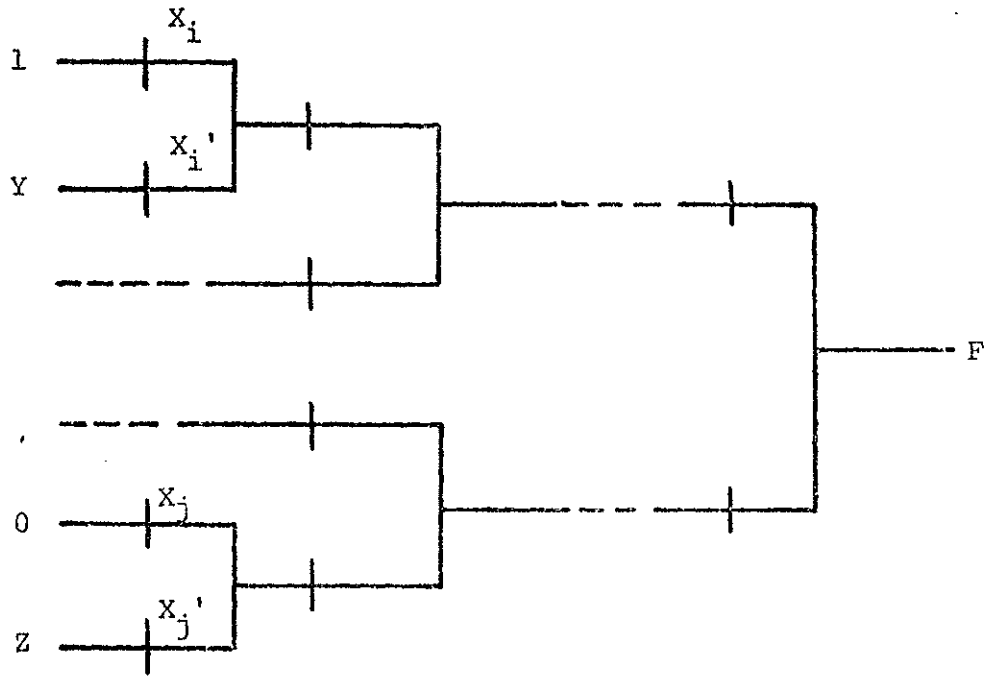


Fig. 6.4. Undetectable S-ON Faults in a BTS Pass Network.

The detection of a s-on fault in a BTS pass network is illustrated by the following example.

Example 6.3: Consider Fig. 6.3 again. A s-on fault on X_{31} can be detected by applying the test vector $\langle X_4X_3X_2X_1\rangle = \langle 1\ 0\ 0\ 1\rangle$. The output $F_1 = 1$ if there is no conflict and $F_1 = 0$ if there is a s-on fault on X_{31}. Similarly, s-on fault test vectors for X_{41}, X_{42}, X_{21}', and X_{22}' can be derived. But s-on fault on X_{41}', X_{31}', X_{42}', and X_{22} cannot be detected since some of the pass variables are constants. The complete set of test vectors for detecting the s-on faults in the different transistors are shown in Table 6.4.

As mentioned earlier for s-op faults, s-on fault test vectors are not unique. Table 6.4 lists only one set of test vectors. Others can be derived in a similar manner.

6.1.3 Internal Node Faults

The faults occurring in the internal nodes of a pass network are classified into s-op, s-at, and bridging faults. Bridging faults occur in integrated circuit chips due to capacitive coupling between neighboring conductors. The effect of these faults depends on the actual patterns laid on silicon. Hence these faults are not considered further in this thesis.

Theorem 6.4: All internal node faults in a BTS pass network are detectable.

Proof: The behavior of s-op faults in the internal nodes of a BTS pass network is similar to s-op faults in

S-on Fault	Test Vector				F_1	F_2
	X_4	X_3	X_2	X_1		
X_{41}'	-	-	-	-	-	-
X_{41}	0	1	0	-	1	-
X_{31}	0	1	0	-	1	-
X_{31}'	-	-	-	-	-	-
X_{21}'	0	-	1	0	-	1
X_{42}'	-	-	-	-	-	-
X_{42}	0	-	0	1	-	1
X_{22}	-	-	-	-	-	-
X_{22}'	1	-	1	1	-	1

Table 6.4. S-on Faults Test Vectors for the Network of Fig. 6.3.

the pass transistor. Hence by Theorem 6.2, all s-op faults in the internal nodes are detectable.

For each internal node there exists at least one path from some pass variable V_i to the output F of the function. Therefore, a s-at fault test for the variable V_i will also test for the same fault in all the internal nodes along that path to the output. Hence a complete set of tests for all the s-at faults in the pass variables will completely test for all the internal node s-at faults. QED.

The test vectors listed in Table 6.1, 6.2, and 6.3 will also detect all s-at and s-op faults in the internal nodes in the network of Fig. 6.3.

The above discussion of fault detection in the BTS pass network has resulted in the following important theorem.

Theorem 6.5: All faults in a BTS pass network are detectable if none of the pass variables are constants.

Proof: The set of faults in a pass network includes s-at faults on pass variables and the internal nodes and s-op and s-on faults in the transistors. Since all these faults are detectable by Theorems 6.1 through 6.4 so long as the pass variables are not constants, all faults are detectable. QED.

6.2 Test Invalidation

The s-on and s-op faults in the transistors merit extra consideration due to the behavior of the pass transistors.

To detect a s-op fault in a transistor a test vector is applied for initialization and then the real test is applied to detect this fault. This scheme does not always gurantee fault detection. Due to unequal delays in the transition of signals between two test vectors, it is possible for charge-sharing to occur between the output node and other circuit nodes via conducting transistors. Once the circuit is initialized, before the application of the real test vector two conditions may occur: (1) All paths to the output get disconnected, and (2) A third path switches the output to the opposite logic state.

If the former condition occurs, then it may happen that the output node no longer holds the right amount of charge necessary for fault detection, thus invalidating the test.

Charge-sharing between the nodes depends on the relative capacitances of the nodes involved. Nodes with capacitances much less than the output node capacitances will probably have significant impact.

An example to illustrate the delay problem occurring in a pass network is given below.

Example 6.4: Consider the detection of s-op fault on transistor X_4' of Fig. 6.5. Assume that the transition from one logic value to another in X_4 is slower than the transition in X_2. To detect this fault use the test vector $\langle X_1\ X_2\ X_4 \rangle = \langle 0\ 1\ 1 \rangle$ as the initialization vector which initializes the output to logic value '1' and then use test

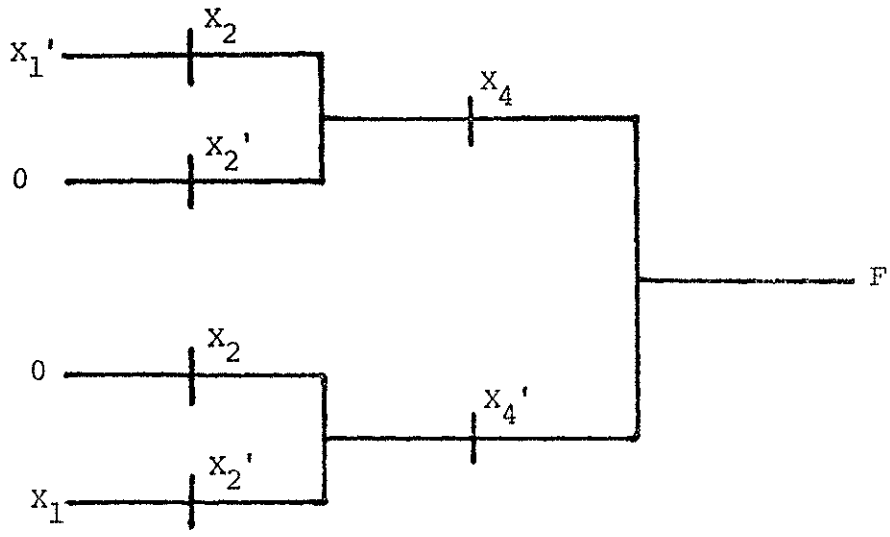


Fig. 6.5. Test Invalidation for Transistor S-OP Fault.

vector $\langle X_1\ X_2\ X_4 \rangle = \langle 0\ 0\ 0 \rangle$ to force the output to logic value '0' if there is no fault. Note that since X_4 is slow in making the transition, a '0' will be passed to the output before the actual test vector is applied thus modifying the initialization to logic '0' and invalidating the test.

The detection of s-on faults in the transistors used the dominance property whereby a '0' dominates a '1'. This may not be valid under all circuit configurations. This happens when a '0' is passed through a longer chain of pass transistors compared to a '1'. Under these conditions the output logic value may be undefined or even switch to logic value '1' thus invalidating the test.

CHAPTER 7

CONCLUSION

The BTS pass network algorithm presented in this thesis has several advantages compared to the earlier ones. It uses at most 2^{n-1} comparisons compared to 2^{n**2} for the optimal algorithm. A tremendous saving in memory requirements is achieved because of the drastic reduction in the number of pass implicants generated.

The BTS pass networks are found to be very suitable for the design of multiple-output pass networks. The necessary and sufficient conditions are derived for sharing part of a network among different pass functions. It is found that this sharing in pass networks is more involved compared to sharing in gate logic networks.

The design of multiple-output pass networks is used to illustrate a new kind of PLA called pass logic PLA. Contrary to our intuition this pass logic PLA is inferior to a normal PLA in terms of the silicon area. But it is superior to gate logic PLA in its speed and also uses less static power. Hence the use of this PLA is dependent on the particular application to which it is used.

Fault detection procedures for multiple-output pass networks are presented with special emphasis given to

multiple-output BTS pass networks. All the faults occurring in these networks are classified into three types: stuck-at faults on pass variables, and stuck-on and stuck-open faults occurring in the pass transistors. The testing of portions of a network which is not shared among different functions is treated similar to single-output functions. Shared network can be tested by observing any one of the outputs which share that portion of the network. It is shown that all stuck-at and stuck-open faults in multiple-output BTS pass networks are detectable. The only problem was with the testing of stuck-on faults. Test vectors are found to be nonexistant for some cases where the pass variables are constants. Hence the test set is found to be incomplete.

Finally, it is shown that some of the test vectors derived earlier for stuck-open and stuck-on faults may become invalid under certain conditions. A stuck-open fault test vector may become invalid due to different delays involved in switching the variables. The assumption of dominance of a '0' over a '1' may invalidate a stuck-on fault if the length of the path feeding a '0' to the output is much longer than that of a '1'path. Hence stuck-on fault test set depends on the actual geometry of the network.

Further work is to be done to find an optimal algorithm for BTS pass networks. A second area for further study must focus on determination of testability criteria for any

network based on its pass function and the topology. This will enable automatic generation of test vectors for any network whenever possible. In addition, it will help to provide design techniques for testable pass networks.

REFERENCE

[1] D. Radhakrishnan, G. K. Maki and S.R. Whitaker, "Formal Design Procedures for Pass Transistor Switching Circuits," IEEE Journal of Solid State Circuits, Vol. sc-20, pp. 531-536, April 1985.

[2] G. E. Peterson and G. K. Maki, "Binary Tree Structured Logic Circuits: Design and Fault Detection," proc. ICCD '84, pp. 671-676, Oct. 1984.

[3] R. L. Wadsack, "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits," The Bell System Technical Journal, Vol. 57, pp. 1449-1474, May-June 1978.

[4] D. Radhakrishnan and G. K. Maki, "Test Derivation for MOS Switch Logic Networks," Proc. 21st Annual Allerton Conf. on Communication, Control, and Computing, pp. 786-795, Oct. 1983.

[5] C. A. Mead and L. A. Conway, Introduction to VLSI systems. Reading, MA: Addison-Wesley, 1980.

[6] H. H. Chen, R. G. Mathews and J. A. Newkirk, "Test Generation for MOS Circuits," proc. IEEE International Test Conference, pp. 70-79, Oct. 1984.

[7] A. R. Feizi and D. Radhakrishnan, "High Performance Switching Circuits for VLSI," Proc. ICCD '85, Port Chester, NY, Oct. 1985.

APPENDIX

PASS PRIME IMPLICANT GENERATION

This is a computer program written in BASIC language to implement the algorithm given in Chapter 3. This program was implemented on a Texas Instruments PC MS-DOS. The program uses two arrays of 2^n and 2^{n-1} sizes (K and M lists). After the first iteration, half of the K-list is used to store the final result (F-list). The pass implicant records are stored in these two arrays during the course of the algorithm. The comparison starts with one array and the resulting pass implicant records are stored in the other array for further comparisons or are stored in the output file as pass prime implicants (subroutine). These two arrays are interchanged during the course of the algorithm till all the pass prime implicants are found which will be stored in the F-list. (pointer S is assigned to the F-list). The given function to be minimized must be in the ascending order and represent 1's of the function (minterm) and must be entered as DATA at the end of the program. When the program is run, the number of minterms, W, and the number of variables, N, must be specified as inputs.

```
10  DIM B(2^12), D(2^12),P(2^12)
20  INPUT "ENTER THE NUMBER OF MINTERMS, W, AND THE NUMBER
    OF VARIABLES, N";W, N
30  IF W<(2N + 1) THEN 60
40  PRINT "INVALID INPUT DATA FOR W &N, TRY AGAIN"
50  GOTO 10
60  PRINT
   'SET THE TIME TO START THE ALGORITHM.
70  TIME$ = "00:00:00"
80  PRINT "W = "W TAB(0) "N = "N
90  A$ = "B(K)       D(K)         P(K)
100  PRINT A$
110  PRINT
     List the given function in the BDP form.
120  FOR J = 0 TO (W - 1)
130  READ B(J)
140  D(J) = 0
150  P(J) = 1
160  NEXT J
      'With the given function, combine two adjacent records
in the list.  At this iteration, all records will combine
because  each  record represent a minterm.  Store the result
in the specified array.
170  I = -2
180  J = 0
190  K = (2^N) - 1
200  B = -2
210  I = I + 2
220  B = B + 2
230  K = K + 1
240  IF B(J) = I AND B(J+1) = I+1 THEN 300
250  IF B(J) = I AND B(J+1) <> I + 1 THEN 330
260  IF B(J) <> I AND B(J+1) <> I + 1 THEN 360
270  P(K) = 11
280  J = J + 1
290  GOTO 380
```

```
300  J = J + 2
310  P(K) = 1
320  GOTO 380
330  J = J + 1
340  P(K) = -11
350  GOTO 380
360  J = J
370  P(K) = 0
380  B(K) = B
390  D(K) = 1
400  IF B(J) <> 0 THEN 210
410  IF B = (2^N) - 2 THEN 540
420  B = B + 2
430  K = K + 1
440  B(K) = B
450  D(K) = 1
460  P(K) = 0
470  GOTO 410
480  FOR K =(2^N) TO ((2^N) + (2^N-1) - 1)
490  PRINT B(K), D(K), P(K)
500  NEXT K
510  FOR K = 0 TO (W-1)
520  PRINT B(K), D(K), P(K)
530  NEXT K
     'Start the comparison with 2^n-1 records in the
      array.  Each records represents two minterms.
540  Z = 1
550  X = 2
560  D = 0
570  M = 0
     'KK determines which array is being compared, K or
      M-list.
580  KK = 0
590  K = 2^N
600  S = ((2^N-1) - 1)
     'Set the D and B-fields for current iteration.
```

```
610  D = D + (2^M)
620  P = -(2Z+1)
630  IF K = 2^N THEN 650
640  GOTO 670
650  J = 0
660  GOTO 680
670  J = 2^N
     'Compare the records K and K+1.
680  IF D(K) <> D THEN 810
690  P = P + (2^Z+1)
700  IF D(K) <> D(K+1) THEN 950
710  Q = P + 2^Z
720  IF B(K) = P THEN 790
730  IF B(K) = Q THEN 760
740  K = K
750  GOTO 680
760  GOSUB 1390
770  K = K + 1
780  GOTO 680
790  IF B(K+1) <> Q THEN 760
800  GOTO 1070
810  IF KK = 0 THEN 830
820  GOTO 850
830  KK = KK + 1
840  GOTO 920
850  KK = KK - 1
860  IF K = 0 THEN 1320
870  K = 2^N
880  Z = Z + 1
890  X = X + 1
900  M = M + 1
910  GOTO 610
920  IF K = 2^N THEN 1320
930  K = 0
940  GOTO 880
950  IF KK = 0 THEN 1010
```

```
960  KK = KK - 1
970  IF K = 0 THEN 1030
980 GOSUB 1400
990  K = 2^N
1000  GOTO 880
1010  KK = KK + 1
1020  IF K <> 2^N THEN 1060
1030  GOSUB 1400
      'End of comparison, print the result.
1040  GOTO 1320
1050  GOSUB 1400
1060  K = 0
1070  GOTO 880
      'Determine if two P-fields can combine.
1080  IF P(K) = P(K+1) = 0 THEN 1180
1090  IF P(K) = 0 AND P(K+1) = 1 THEN 1240
1100  IF P(K) = 1 AND P(K+1) = 0 THEN 1260
1110  IF P(K) = P(K+1) = 1 THEN 1280
1120  IF P(K)=P(K+1) <> 1 AND P(K)=P(K+1) <> 0 THEN 1300
      'The records do not combine, send to the output file.
1130  GOSUB 1400
1140  K = K + 1
1150  GOSUB 1400
1160  K = K + 1
1170  GOTO 680
1180  P(J) = 0
1190  D(J) = B(K+1) - B(K) + D(K)
1200  B(J) = B(K)
1210  K = K + 2
1220  J = J + 1
1230  GOTO 680
1240  P(J) = X
1250  GOTO 1190
1260  P(J) = -X
1270  GOTO 1190
1280  P(J) = 1
```

```
1290  GOTO 1190
1300  P(J) = P(K)
1310  GOTO 1190
1320  FOR K = (2^N-1) TO S
1330  PRINT B(K), D(K), P(K)
1340  NEXT K
1350  PRINT
      'Total time to generate the pass prime implicants.
1360  PRINT "COMPUTATION TIME TC = "TIME$
1370  DATA 0,1,4,5,11,12,14
1380  END
      SUBROUTINE TO STORE THE FINAL RESULT
1400  S = S + 1
1410  B(S) = B(K)
1420  D(S) = D(K)
1430  P(S) = P(K)
1440  RETURN
```

The final result is in BDP form and must be converted to BTS form using the procedure given in section 3.3. The P-field a 1, 0, or numbers 11, 2, 3, 4,..., corresponding to variables X_1, X_2, X_3, X_4,... respectively. A minus sign is used to represent complementry variables. For example, if P-field contains 1, -11, 11, -3, -8, 7, then the coresponding pass variables are: 1, X_1', X_1, X_3', X_8', X_7 respectively.

The D-field in the final result obtained from the above program is equal to the sum of the terms appearing in the D-field, d, and must be converted to $(2^0,2^1,2^2,\ldots,2^{n-1})$ form because $d = 2^0+2^1+2^2+\ldots+2^{n-1}$. The following examples are used to illustrate the implementation of the above program.

Example 1: $f(X_3\ X_2\ X_1) = m(0,2)$.

```
DATA  0,2
RUN
ENTER  THE  NUMBER OF MINTERMS W AND THE NUMBER OF VARIABLES
N? 2,3
W = 2
N = 3
```

PRINT-OUT

B(K)	D(K)	P(K)
0	3	-11
0	3	0

CONVERSION

B(K)	D(K)	P(K)
0	(1,2)	X_1'
0	(1,2)	0

EXAMPLE 2: $f(X_4\ X_3\ X_2\ X_1) = (0,1,4,5,9,11,14)$

```
DATA 0,1,4,5,9,11,14
RUN
ENTER THE NUMBER OF MINTERMS W AND THE NUMBER  OF  VARIABLES
N? 8,4
W = 8
N = 4
```

PRINT-OUT

B(K)	D(K)	P(K)
8	3	11
12	3	-11
0	7	-2

CONVERSION

B(K)	D(K)	P(K)
8	(1,2)	X_1
12	(1,2)	X_1'
0	(1,2,4)	X_2'

Example 3: $f(X_4\ X_3\ X_2\ X_1) = (3,7,12,13,14,15)$

```
DATA 3,7,12,13,14,15
RUN
```

ENTER THE NUMBER OF MINTERMS W AND THE NUMBER OF VARIABLES

N? 6,4

W = 6

N = 4

PRINT-OUT

B(K)	D(K)	P(K)
0	1	0
2	1	11
4	1	0
6	1	11
8	7	3

CONVERSION

B(K)	D(K)	P(K)
0	1	0
2	1	X_1
4	1	0
6	1	X_1
8	(1,2,4)	X_3

HIGH PERFORMANCE SWITCHING CIRCUITS FOR VLSI

by

Ali Reza Feizi

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF ENGINEERING

ELECTRICAL ENGINEERING

OLD DOMINION UNIVERSITY
December 1985

ABSTRACT

HIGH PERFORMANCE SWITCHING CIRCUITS FOR VLSI

Ali Reza Feizi
Old Dominion University, 1985
Director: Dr. S.V. Kanetkar

Interconnection topology and device performance are of major concern in the design of LSI/VLSI systems. Pass networks are very suitable in this regard because of low power consumption, high density, and simple interconnection topology. A special type of pass networks called Binary Tree Structured (BTS) pass networks uses almost minimum number of transistors for the design of switching circuits. An algorithmic procedure is developed here for BTS pass networks which is very efficient in terms of both excution time and memory space. Based on these networks, the necessary and sufficient conditions are derived for the design of multiple-output pass networks. A new kind of Programmable Logic Arrays (PLA) called pass logic PLA is also proposed in this thesis. Fault detection techniques for multiple-output pass networks with special attention given to multiple-output BTS pass networks are presented here. Finally, it is found that under certain conditions some of the faults which occur in pass networks cannot be detected.