

THRESHOLD ELEMENTS AND THE DESIGN OF
SEQUENTIAL SWITCHING NETWORKS

Massachusetts Institute of Technology
Electronic Systems Laboratory

This document is subject to special
export controls and each transmittal
to foreign governments, foreign na-
tionals or representatives thereto may
be made only with prior approval of
RADC (EMIIO), GAFB, N.Y. 13440.

FOREWORD

This final report was prepared by the Massachusetts Institute of Technology, Electronic Systems Laboratory, Cambridge, Massachusetts, under contract AF 30(602)-3681, Project 5581, Task 09. MIT Project No. is DSR 74631; the secondary report number is ESL-FR-354.

This report covers the period of work from March 1967 to June 1968. The RADC Project Engineer is Mr. James Previte, EMIIO.

The following MIT personnel participated in the conduct of the Project:

Research Staff

Prof Alfred K. Susskind, Project Engineer
Dr. Donald R. Haring
Robert L. Martin
Charles Ying
Richard Diephuis
Satish Gupta
John M. Mazola

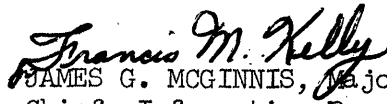
Thesis Students

David G. Chapman
William M. Inglis
Nicholas J. Pippenger

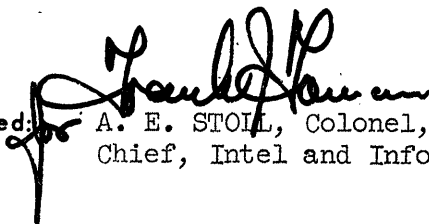
The distribution of this document is limited under the U.S. Mutual Security Acts of 1949.

This report has been reviewed and is approved.


Approved:


JAMES G. MCGINNIS, Major, USAF
Chief, Information Processing Branch

Approved:


A. E. STOLL, Colonel, USAF
Chief, Intel and Info Processing Division

FOR THE COMMANDER:


IRVING J. GABELMAN
Chief, Advanced Studies Group

ABSTRACT

This report on research performed from March 1967 to June 1968 under Contract AF-30(602)-3681 is divided into six sections. Section I is an introduction. Section II is concerned with the realization of sequential machines in the form of a bidirectional shift register and in the form of a unidirectional shift register augmented with a shift register that stores inputs. This section also treats techniques for making sequential machines diagnosable and decomposition of sequential machines with one submachine in the form of a clock. Section III deals with fault detection in combinational logic for (a) arbitrary networks with single faults, (b) single threshold gates with multiple faults, and (c) fault detection and location based on fault tables. Section IV gives further results in the synthesis of multivalued logic networks. Section V reports on a study of two classes of module functions as building blocks in a digital system. The final section discusses topics in design automation of digital systems.

TABLE OF CONTENTS

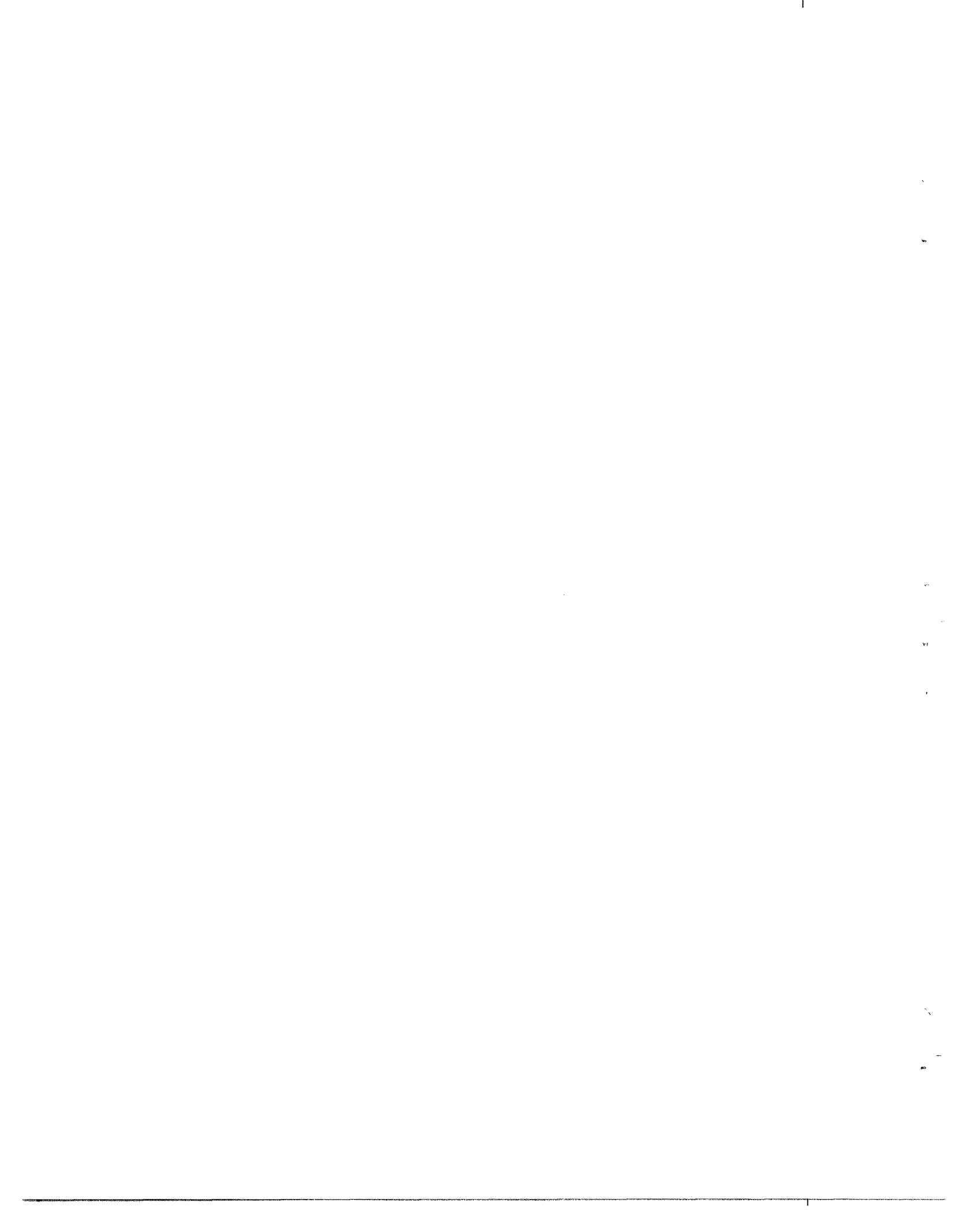
	<u>Page</u>
SECTION I INTRODUCTION	1
SECTION II FEEDBACK SHIFT-REGISTER RELATED TOPICS	8
CHAPTER 1 SYNTHESIS OF SEQUENTIAL MACHINES IN THE FORM OF A FEEDBACK SHIFT-REGISTER AUGMENTED BY AN INPUT- DRIVEN SHIFT REGISTER	9
1.1 THE ASSUMED CIRCUIT FORM	9
1.2 BACKGROUND	11
1.3 THE SYNTHESIS TECHNIQUE	13
1.4 REDUCING THE REGISTER LENGTHS	18
1.5 A HEURISTIC TECHNIQUE	25
CHAPTER 2 FEEDBACK SHIFT-REGISTERS AND THE DIAGNOSIS OF SEQUENTIAL MACHINES	30
2.1 DIAGNOSING FSR REALIZATIONS	33
2.2 FLOW-TABLE MODIFICATION BY COLUMN ADDITION	38
2.3 FLOW-TABLE MODIFICATION BY OUTPUT ADDITION	41
2.4 SUMMARY	45
REFERENCES	46
CHAPTER 3 BI-DIRECTIONAL FEEDBACK SHIFT-REGISTER SYNTHESIS OF SEQUENTIAL MACHINES	47
3.1 THE SYNTHESIS ALGORITHM	49
3.2 EQUIVALENT STATES	54
REFERENCES	58
CHAPTER 4 REALIZATION OF SEQUENTIAL MACHINES IN A CIRCUIT FORM CONTAINING A CLOCK	59
4.1 STATE DIAGRAM OF A CLOCK	60
4.2 TEST FOR EXISTENCE OF A CLOCK	61

TABLE OF CONTENTS (Continued)		<u>Page</u>
4.3	AN ASIDE REGARDING FSR SYNTHESIS	64
4.4	USE OF EQUIVALENT STATES IN DECOMPOSITION	64
4.5	CONCLUSIONS	71
	REFERENCES	72
SECTION III	FAULT DETECTION AND LOCATION IN COMBINATIONAL LOGIC	73
CHAPTER 1	AN ALGORITHM FOR FINDING APPROXIMATELY MINIMAL FAULT-DETECTION TEST SETS FOR COMBINATIONAL LOGIC	74
1.1	ASSUMPTIONS	75
1.2	PATH SENSITIZING	75
1.3	NOTATION	81
	1.3.1 Specification of the Network Behavior - The Singular Cover	81
	1.3.2 Representation of a Sensitized Path - D-Cubes	84
	1.3.3 Primitive D-Cubes	89
	1.3.4 Representation of a Test - T-Cubes	91
	1.3.5 The Operation of Intersection	92
1.4	CONSTRUCTION OF D-CUBES FOR SENSITIZED PATHS OF A NETWORK	97
	1.4.1 Formation of Incomplete D-Cubes	97
	1.4.2 Completing the D-Cubes	98
1.5	FORMATION OF TESTS FROM COMPLETED D-CUBES	101
	1.5.1 Formation of Primitive T-Cubes from Completed D-Cubes	102
	1.5.2 Formation of Composite T-Cubes from Primitive T-Cubes	104
1.6	ALGORITHM FOR FINDING AN APPROXIMATELY MINIMAL TEST SET FOR A GIVEN NETWORK	106

TABLE OF CONTENTS (Continued)		<u>Page</u>
1.7	FAULTS WHICH CANNOT BE DETECTED BY SENSITIZING SINGLE PATHS FROM AN INPUT TO AN OUTPUT	107
1.7.1	Class 1	108
1.7.2	Class 2; Sensitizing Multiple Paths	108
1.7.3	The Modified Algorithm	111
1.8	CONCLUSIONS AND DISCUSSION	112
	REFERENCES	115
APPENDIX 1.1	NETWORKS WITH FAULTS FOR WHICH TESTS CANNOT BE FOUND BY THE METHOD DESCRIBED	116
CHAPTER 2	CHECKING OF SINGLE THRESHOLD ELEMENTS	119
2.1	THE FAILURE MODEL	119
2.2	UNATENESS AND POSITIVE WEIGHTS	121
2.3	PRELIMINARIES	122
2.4	ALGORITHM FOR FINDING SHORT TEST SETS	127
2.5	BOUNDS ON THE LENGTH OF THE CHECKING SET	137
2.6	FINDING TEST SEQUENCES FOR RELATED FUNCTIONS	140
	REFERENCE	147
APPENDIX A	UPPER BOUND ON THE NUMBER OF PRIME IMPLICANTS OF UNATE FUNCTIONS	148
CHAPTER 3	AN ALTERNATIVE APPROACH TO CHECKING BASED ON A RESPONSE-TABLE	154
3.1	ILLUSTRATIONS OF APPROACH	155
3.2	FINDING SHORTEST CHECKING TESTS	160
3.3	FINDING DIAGNOSING TESTS	163
3.4	CONCLUSIONS	166
SECTION IV	MULTI-VALUED LOGIC	168
CHAPTER 1	INTRODUCTION	169

TABLE OF CONTENTS (Continued)		<u>Page</u>
CHAPTER 2	THE MUHLDORF SYSTEM OF MULTI-LEVEL SWITCHING	173
CHAPTER 3	SYMMETRY IN M-VALUED FUNCTIONS	188
3.1	DEFINITIONS OF SYMMETRY IN M-LEVEL FUNCTIONS	189
3.2	PROPERTIES OF SYMMETRIC FUNCTIONS	191
3.3	ALGORITHM TO DETECT SYMMETRIC FUNCTIONS	209
	REFERENCES	215
SECTION V	MODULE SYNTHESIS OF SWITCHING FUNCTIONS	217
CHAPTER 1	INTRODUCTION	218
1.1	THE M_1 MODULE	218
1.2	OUTLINE OF THIS REPORT	219
CHAPTER 2	THE M_2 MODULE	221
2.1	INTRODUCTION TO THE M_2 MODULE	221
2.2	COMPLETENESS AND ASYMMETRY OF M_2^n	222
2.3	LOGIC COMPLEXITY OF M_2^n AND M_1^n	224
2.4	SEARCH OF ALL THREE-VARIABLE FUNCTIONS FOR MODULE FUNCTIONS	236
CHAPTER 3	OPERATIONS ON M_1^n AND M_2^n	240
3.1	BIASING AND DUPLICATING OPERATIONS	240
3.2	SINGLE-ELEMENT-REALIZABLE FUNCTIONS	242
3.3	STANDARDIZATION OF A WELL-ORDERED SEQUENCE	244
3.4	REALIZATION PROCEDURE	245
3.5	REALIZATION OF EX-OR, AND, and OR FUNCTIONS	250
3.6	REALIZATION OF M_2^{n-1} AND M_1^{n-1} FROM M_2^n AND M_1^n	252
3.7	REALIZATION OF M_2^{n+1} AND M_1^{n+1} FROM M_1 AND M_2 MODULES WITH AT MOST n PINS	253

TABLE OF CONTENTS (Continued)		<u>Page</u>
CHAPTER 4	SYNTHESIS BY M_2	256
4.1	A TWO-LEVEL SYNTHESIS APPROACH	256
4.2	TREE SYNTHESIS APPROACH	258
CHAPTER 5	SYNTHESIS OF ALL THREE-VARIABLE SWITCHING FUNCTIONS	265
CHAPTER 6	SYNTHESIS OF FOUR-VARIABLE FUNCTIONS	271
6.1	CASCADE OF TWO M_1 MODULES	272
6.2	CASCADE OF TWO M_2 MODULES	281
CHAPTER 7	CONCLUSIONS	290
	REFERENCES	292
APPENDIX A	SOME DECOMPOSITIONS OF SWITCHING FUNCTIONS	293
SECTION VI	DESIGN AUTOMATION FOR DIGITAL SYSTEMS	300
CHAPTER 1	OVERVIEW	301
1.1	THE DATA STRUCTURE DESCRIBING THE DESIGNED SYSTEM	302
1.2	THE DATA STRUCTURE DESCRIBING AVAILABLE PRINTED CIRCUIT BOARDS	305
1.3	ASSIGNMENT OF DESIGNED CIRCUITRY TO AVAILABLE CIRCUIT BOARDS	305
1.4	PLACEMENT OF PC BOARDS ON A BACKPLANE	309
1.5	DETERMINATION OF OPTIMUM WIRE ROUTING	310
1.6	SIMULATION OF THE DESIGNED SYSTEM	311
1.7	AUTOMATIC PREPARATION OF LOGIC DIAGRAMS	314
CHAPTER 2	AN ASSIGNMENT ALGORITHM BASED ON LINEAR PROGRAMMING	318
2.1	A COMPARISON-ORIENTED DATA STRUCTURE	318
2.2	COMPARISON OF CANONICAL GATING EXPRESSIONS	321
2.3	THE OPTIMIZING ASSIGNMENT ALGORITHM	322



SECTION I

INTRODUCTION

A. K. Susskind

This is the final report published under Contract AF30(602)-3681. It covers in considerable detail most of the work completed since March 1967.

Since its inception in March 1965, the major motivation for the work under this contract has come from the demands brought about by current trends in device and circuit technology. With the advent of complex integrated circuits, emphasis in logic design has shifted away from attempting to minimize the number of nonlinear elements, because their cost is no longer the dominant consideration. Rather, one wants to minimize the variety of building blocks and their total number, where now the complexity of the building block (i.e., the number of nonlinear elements that it contains) is a matter of ever decreasing concern. Furthermore, one desires to make use of computers in the design process in order to minimize design tedium and cost, as well as to increase assurance that the design will operate as desired. Finally, there is a need for efficient techniques for checking and diagnosing the sealed boxes which now make up the constituents of a digital system, and there is every reason to expect that the functional power of the individual boxes will continue to grow with time.

We find conventional logic design ill equipped to accommodate the shift of interest. The time is approaching when it will no longer be satisfactory to specify a design in terms of conventional logic gates, such as NAND, and flip-flops. But with what building blocks shall these be replaced? And what are efficient design techniques for exploiting the replacements to their fullest? These are among the key issues to which our work has addressed itself.

It is well known that, because they have greater logic power, threshold elements take building blocks that can, for a given logic function, significantly reduce the building-block count over what it would be with conventional gates. In our first report of the same title (Technical Report No. RADC-TR-66-286), we introduced a novel extension of the threshold element, the so-called multi-threshold threshold element (MTE). We showed that a single MTE can realize any single-output switching functions. Hence it is the most powerful single-output gate possible. We described how MTE realizations of functions can be specified, how they can be translated into networks of threshold elements, and finally we showed some of the many possible electrical circuit forms that one could use. In our second report of the same title (Technical Report No. RADC-TR-67-255), we extended our investigation of MTE logic along two lines. First, we presented a tabular synthesis procedure which, with little computational complexity, allows one to find a weight-threshold vector that can realize any function of four or less variables. The resulting realizations are in general very good, and in many cases they are optimum. The second extension presented a novel method for MTE synthesis, well suited to execution by a digital computer. A program using this method was run on the M.I.T. time-sharing system and its performance was excellent. Furthermore, the concepts involved in this new method can be applied to other problems of interest, such as single-threshold synthesis, multiple-output synthesis, and multi-level logic.

In the area of sequential machines, a large part of our effort has centered on synthesis techniques for realizations in the form of a shift register. This circuit form is particularly well suited to the new

technology because the major functional block, i.e., the shift register, is in fixed form for all possible machines that are shift-register realizable, and only the length of the register can differ for different machines. Indeed, shift registers in integrated circuit form are now commercially available in compact form and at low cost. Furthermore, the techniques of shift register synthesis are also applicable to the case where the variety of signals on a lead is greater than two. Thus the shift-register form offers one approach to exploiting in sequential machines logic with more than two levels, should such logic become practicable. Finally, the shift register form permits the inclusion of error control with only a modest increase in circuitry.

In our first report, we presented a novel algebraic technique for shift-register synthesis. This technique was extended in our second report along two directions: synthesis for the case of multiple registers and synthesis for the case where simple logic can be inserted between the cells of the shift register. The results of our work are design algorithms and, as is so often the case with design algorithms, they require some trial and error. To expedite the design process, we wrote a rather general computer programming system, also described in our second report. This system has a very simple problem-oriented command language and has been used successfully in shift-register design. Our second report also introduced a new approach to the synthesis of sequential machines in shift-register form. Here the key concepts were memory span and the pair graph. What makes the new approach so powerful is its ability to deal with state assignments that permit more than the minimum number of state variables through the creation of equivalent states by an implicit mechanism.

In this report, further extensions of the work in shift-register synthesis are given in Section II. First, we discuss the realization of a sequential machine in the form of a feedback shift-register augmented with a register that stores inputs only. The advantage of this form lies in the fact that it can realize every machine, whereas without augmentation the circuit form is not universal. Section II also discusses techniques for making sequential machines diagnosable, and while these techniques are applicable to all sequential machines, no matter what the form, their foundation lies in the concepts which were generated in our studies of shift-register synthesis. Chapter 3 of Section II discusses synthesis in the form of a variant of the feedback shift-register, where information is shifted either right or left. The final chapter in this section is concerned with decomposition of sequential machines where one component is an input independent, and hence shift-register realizable, machine.

Section III of this report deals with fault detection and location in combinational logic. Here three studies are reported on. The first is concerned with checking of arbitrary networks consisting of conventional logic (AND, OR, NAND, NOR) and presents new techniques for finding short tests that determine whether or not the network on hand does in fact operate as desired, assuming that only a single fault is present. The second chapter is concerned with checking of threshold elements under the assumption that arbitrarily many faults can exist. Section III ends with an approach to finding sets of tests for both diagnosis and checking, assuming that a fault table is given.

One way of increasing the logic power of digital building blocks is to permit each signal lead to carry more than two signal levels. This leads to multilevel logic. Our first results in this area were given in RADC-TR-67-255, where we showed several generalizations and extensions of older concepts and introduced several concepts that are analogous to conventional majority logic and conventional threshold logic. A small effort, not reported on, showed that circuit realizations of multilevel logic are feasible, given that electrical circuit complexity is no longer of paramount importance. In this report, Section IV gives further theoretical results in the area of multilevel logic and discusses in detail a synthesis technique based on electrically simple circuit blocks.

Several other approaches toward the specification and exploitation of more powerful logic blocks were also undertaken. In our first technical report (RADC-TR-66-286) we discussed novel multiple-input, multiple-output boxes of moderate complexity and showed how they can be used to realize ensembles of Boolean functions. It is noteworthy that the synthesis algorithm involves no trial-and-error, that the language used by the designer is not Boolean, and that the algorithm can handle any number of Boolean functions not exceeding the number of inputs. In our next report (RADC-TR-67-255) we discussed synthesis with universal logic blocks, by which we mean networks that, by means of their external connections only, can be made to realize any one Boolean function. In this report, further work with powerful logic blocks is documented. Section V treats two classes of functions that have simple algebraic structure. While ad-hoc synthesis techniques are given, we were not able to penetrate deeply enough to find fully adequate design concepts for the exploitation of the two classes of functions.

The final section of this report is concerned with the field of design automation of digital systems. It contains a new approach to the efficient assignment of printed circuit boards to system function.

In this report, as in its predecessors, the number of topics covered is rather large. Consequently, it has not been possible to provide with each chapter an adequate treatment of the appropriate background. However, an effort was made throughout to list suitable references, and it is hoped that these will assist the reader in gaining access to recent developments in the field.

SECTION II

FEEDBACK SHIFT-REGISTER RELATED TOPICS

This section reports on several studies. Chapter 1 (by R. L. Martin) deals with the synthesis of sequential machines in the form of a feedback-shift-register augmented by a shift-register that stores inputs only. Chapter 2 (by R. L. Martin) deals with techniques for making sequential machines efficiently diagnosable so that short checking experiments can be found. The new techniques introduced are a direct product of previous studies in shift-register synthesis. Chapter 3 (also by R. L. Martin) describes a novel approach to the synthesis of a sequential machine in the form of a register that can shift both right and left. Finally, Chapter 4 (by A. K. Susskind) shows how the concepts of shift-register synthesis can be effectively utilized to determine if a given sequential machine can be decomposed into two submachines, one of which is input independent.

CHAPTER 1

SYNTHESIS OF SEQUENTIAL MACHINES IN THE FORM OF A FEEDBACK SHIFT-REGISTER AUGMENTED BY AN INPUT-DRIVEN SHIFT REGISTER

In Section 3.4 of our previous report (Ref. 1) we introduced the relationship between finite input memory span and feedback shift-register (FSR) realizability. In particular, we defined a set of rules by which a flow table can be "transformed" and showed that a sequential machine (SM) with no inaccessible states is L-level FSR realizable iff it has a L-column transformed flow table having finite input-memory span (Theorem 3.4). Further studies along the lines of finite input memory span were conducted. A number of important results were obtained, and these will be contained in a monograph to be published by the M.I.T. Press. Unfortunately, this work did not lead to fully satisfactory synthesis techniques. Here we present an extension of this work which leads to a very appealing and useful method of synthesis.

1.1 THE ASSUMED CIRCUIT FORM

Whereas before we assumed that the SM machine is to be realized by an FSR with L levels and combinational logic only, we now add a shift register that stores inputs only. The resultant circuit form is shown in Fig. 1.1, where the upper row of delays represents the shift-register that stores inputs and the lower row the FSR. Note that the feedback logic has as its input at time t primary inputs $I(t), I(t-1), \dots, I(t-p)$, where p is the length of the shift register. It also has as inputs all the state variables of the present state stored in the FSR. The logic shown feeds only the FSR. The SM outputs to the "outside world" are not

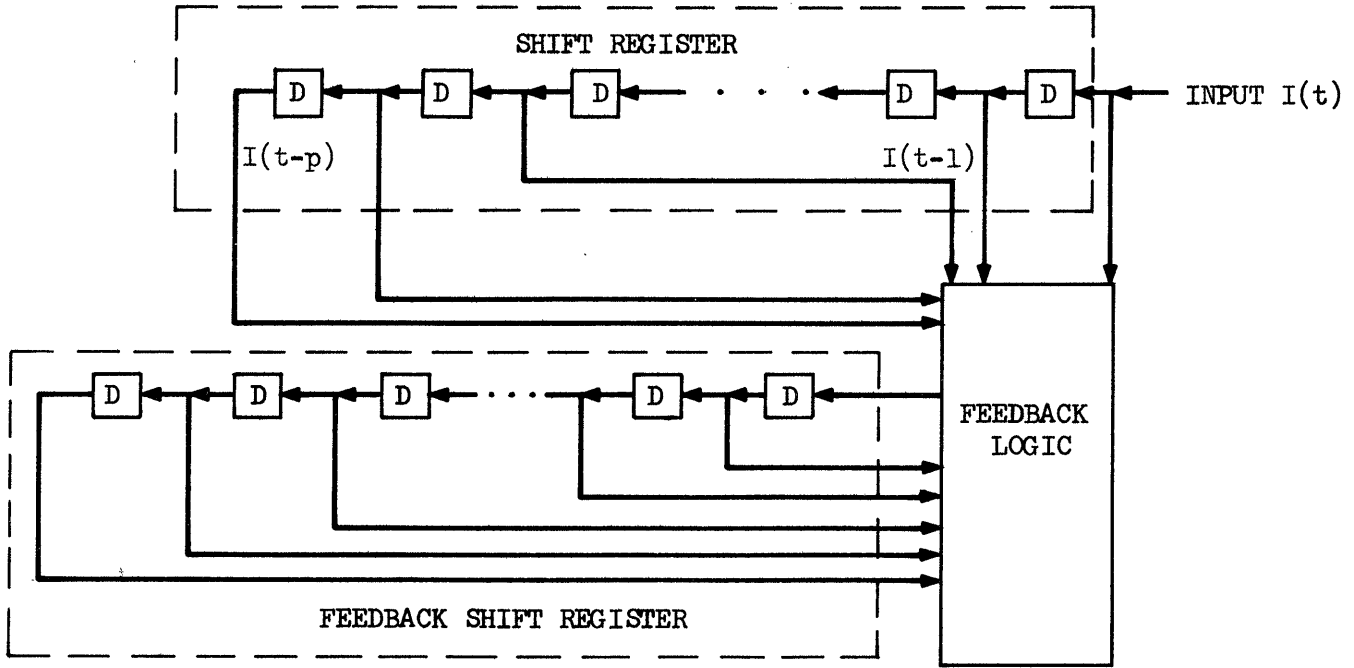


Fig. 1.1 Assumed Circuit Form

shown. They are generated by a separate logic block not included in Fig. 1.1, because we are interested only in realizing the state behavior.

That any SM can be realized in the form of Fig. 1.1 was shown by Friedman (Ref. 2), who also showed that a two-level ($L = 2$) FSR is always sufficient. However, the number of register stages required can be prohibitively large when $L = 2$. Here we allow L to be larger than two and consequently obtain a circuit that is frequently (but not always) attractive from the economic point of view.

1.2 BACKGROUND

The concept of the "output transformed table" will be useful in this chapter. From a flow table with outputs assigned to successors of states, the output transformed table is generated such that if, under input I_s state S_i maps to S_j with output N_i , then in the output transformed table S_i maps to S_j under input I_s, N_i . For example, the output transformed table of the flow table of Figure 1.2a is shown in Figure 1.2b.

From the discussion concerning FSR realizations of SM's in Ref. 1, it is easily seen that:

Lemma 1.1: A SM can be realized with an input-fed shift register in parallel with an output-fed shift register iff its output transform has finite memory span*.

For example, the output transform of Figure 1.2b has length three memory-span; hence, it can be realized with a three-stage, output-fed,

* We can say "iff" here as the output transformed table must have don't-cares in it; hence, there is no problem in generating assignments for the transient states of a flow table.

	0	1
1	1,0	2,1
2	4,1	5,0
3	1,1	3,0
4	3,0	4,1
5	3,1	4,0

(a)

	0,0	0,1	1,0	1,1
1	1	-	-	2
2	-	4	5	-
3	-	1	3	-
4	3	-	-	4
5	-	3	4	-

(b)

Figure 1.2 A SM and Its Output Transformed Table

binary FSR in parallel with a three-state, input-fed, binary shift register.

We shall now determine how to append outputs to a flow table (or more precisely a state table) so that its output transform has finite memory span. Our approach to this problem is similar to that of Kohavi and LaVallee (Ref. 3) in their study of definitely diagnosable SM's. After outlining their approach, we shall extend it and find an algorithm which can be used to determine the minimum number of output levels that one must add to a flow table so that its output transform has finite memory span; that is, we wish to determine that L such that an L-valued, but not (L-1)-valued, output can be added to a flow table so that its output transform has finite memory.

As an aid in explaining this approach, consider the flow table of Figure 1.3a. Under input restriction alone, Figure 1.3b represents the mapping properties of all the pairs of states of the SM. (Again, a graph of this form is called an input pair-graph.)

	0	1
1	1	2
2	3	1
3	5	3
4	2	4
5	1	2

Figure 1.3(a) Example Machine

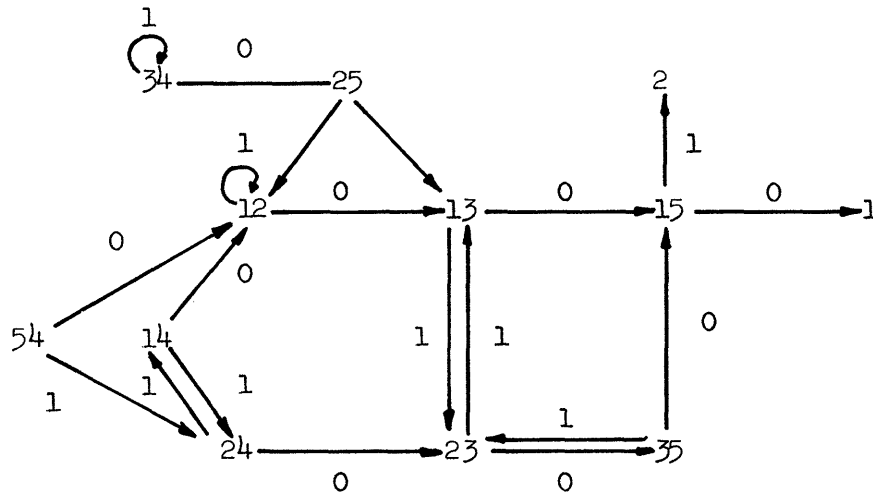


Figure 1.3(b) Input Pair-Graph

As Kohavi and LaVallee indicate, the input pair-graph of the output transform of the SM of Figure 1.3a will have finite memory iff all the cycles in the graph of Figure 1.3b are eliminated. We shall call the input pair-graph of the output transform an output pair-graph.

1.3 THE SYNTHESIS TECHNIQUE

A cycle of pairs of states of length r in an output or input pair-graph will be represented by an r -element vector. The vector $(35^1, 23^0)$ represents the following cycle in Figure 1.3b: 35 mapping to 23 under input 1, followed by 23 mapping to 35 under input 0. To make Figure 1.3b cycle free, it is necessary and sufficient to add outputs to the

states so that only those cycles in which a node appears once are eliminated. This will automatically eliminate those cycles in which a node appears twice, such as $(13^1, 23^0, 35^1, 23^1)$.

In general, any cycle $((S_{i1}S_{j1})^{I_1^r}, (S_{i2}S_{j2})^{I_2^r}, \dots, (S_{ir}S_{jr})^{I_r^r})$ can be broken by adding different outputs to at least one state pair (S_{ip}, S_{jp}) under input I_p^r . For example, the cycle $(35^1, 23^0)$ can be eliminated by either adding different outputs to states 2 and 3 under input 0, or different outputs to states 3 and 5 under input 1.

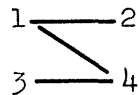
The condition for elimination of a single cycle can be easily represented by the Boolean OR, i.e., different outputs must be added to the states $S_{i1}S_{j1}$ under input I_1^r , OR ..., OR to the states $S_{ir}S_{jr}$ under input I_r^r . But all cycles must be eliminated for the output pair-graph to have finite memory. This condition can be represented by the Boolean AND. In other words, if propositions C_1, C_2, \dots, C_q represent the various ways in which cycles 1, 2, ..., q are broken, then the Boolean product $(C_1)(C_2)\dots(C_q)$, multiplied out in sum-of-products form, represents the various ways in which all cycles can be broken. Let $(ij)^k$ represent the proposition "the outputs of states i and j under input k are made different". For the graph of Fig. 1.3b we obtain

$$\begin{aligned} C_1 &= (12)^1 \\ C_2 &= (14)^1 + (24)^1 \\ C_3 &= (34)^1 \\ C_4 &= (13)^1 + (23)^1 \\ C_5 &= (23)^0 + (35)^1 \end{aligned}$$

$$\begin{aligned}
 P &= C_1 C_2 C_3 C_4 C_5 \\
 &= (12)^1 [(14)^1 + (24)^1] (34)^1 [(13)^1 + (23)^1] [(23)^0 + (35)^1] \\
 &= (12)^1 (34)^1 (14)^1 (13)^1 (23)^0 + (12)^1 (34)^1 (14)^1 (13)^1 (35)^1 + \\
 &\quad (12)^1 (34)^1 (14)^1 (23)^1 (23)^0 + (12)^1 (34)^1 (14)^1 (23)^1 (35)^1 + \\
 &\quad (12)^1 (34)^1 (24)^1 (13)^1 (23)^0 + (12)^1 (34)^1 (24)^1 (13)^1 (35)^1 + \\
 &\quad (12)^1 (34)^1 (24)^1 (23)^1 (23)^0 + (12)^1 (34)^1 (24)^1 (23)^1 (35)^1
 \end{aligned}$$

Hence, either different outputs must be added to 1 and 2 under 1; 3 and 4 under 1; 1 and 4 under 1; 1 and 3 under 1; and 2 and 3 under 0; or different outputs must be added to 1 and 2 under 1; 3 and 4 under 1; 1 and 4 under 1; 1 and 3 under 1; and 3 and 5 under 1; or ...; different outputs must be added to 1 and 2 under 1; 3 and 4 under 1; 2 and 4 under 1; 2 and 3 under 1; and 3 and 5 under 1.

Now we shall determine the minimum variety of outputs one must add so that one of these expressions is satisfied. First, we determine the minimum variety one must add so that all the links indicated by the first product are broken. For this it is helpful to construct an adjacency map in which state i is adjacent to state j in the map for input k iff the term $(ij)^k$ is present in the first product term. Hence the adjacency map for input 1 of the first term is



and for input 0 it is



Adjacent nodes on a map must have different outputs assigned to them. Since the variety of outputs determines the number of levels (L)

in the FSR, one wants to determine the minimum variety of outputs necessary for a given map. As noted by F. C. Hennie, we are faced with the classical map-coloring problem. Though no graphical solution to this problem is known, linear programming or trial and error can be used to find the minimum number of colors (levels) for a particular map. The above map for input 1 requires 3 levels, while the map for input 0 requires two levels. Hence, if the first term is chosen for breaking all cycles, then we would have to add a three-valued output to transform the table so that its output transform has finite memory. However, the maps for $(12)^1 (34)^1 (14)^1 (23)^1 (23)^0$, $(12)^1 (34)^1 (14)^1 (23)^1 (35)^1$, $(12)^1 (34)^1 (24)^1 (13)^1 (23)^0$, and $(12)^1 (34)^1 (24)^1 (13)^1 (35)^1$ require only two levels. Hence only a single binary output must be added to the SM of Figure 1.3a so that its output transform has finite memory.

Choosing the term $(12)^1 (34)^1 (14)^1 (23)^1 (23)^0$, we find the following adjacency graphs. The outputs assigned as shown in Figure 1.3d break all links in these two graphs; hence, the output transform of the SM of Figure 1.3d has finite memory regardless of the specification of the remaining don't-cares.

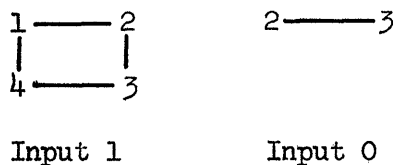


Figure 1.3(c) The Adjacency Graphs

	0	1
1	1/∅	2/0
2	3/1	1/1
3	5/0	3/0
4	2/∅	4/1
5	1/∅	2/∅

Figure 1.3(d) Example Machine with Outputs Added

The following algorithm is a summary of the preceding procedure and will result in the minimum variety of outputs which must be added to a SM so that its output transform has finite memory.

Algorithm 1.1:

1. Form the input pair-graph of the SM. If S_i and S_j map to S_i^q and S_j^q , respectively, under input q , then $S_i S_j$ maps to $S_i^q S_j^q$ under input q in the input pair-graph.
2. Find all cycles in the input pair-graph and express the condition for eliminating the r^{th} cycle as $C_r = (S_{i1} S_{j1})^{I_1} + (S_{i2} S_{j2})^{I_2} + \dots + (S_{ir} S_{jr})^{I_r}$.
3. Form the Boolean product of all the C_i and express the result in sum-of-products form.
4. For each product term Q in the sum-of-products expansion, form an adjacency graph for each input used as a superscript in one of the elements in Q . If $(S_{i1} S_{j1})^q$ is present in Q , then in the q adjacency graph for Q connect S_{i1} to S_{j1} by a branch.
5. For each set of adjacency maps, by using map-coloring techniques, determine the minimum variety of outputs, M_i , in order that outputs can be assigned to the nodes of each map in the set so that adjacent nodes have different outputs. Let M_m

denote the smallest M_1 over all sets of adjacency maps. Assign M_m -valued outputs to the states of the SM so that adjacent nodes in the set of maps which gave rise to M_m have different outputs. Assign an output to a state of the SM under input q only if it is present in the q adjacency map.

1.4 REDUCING THE REGISTER LENGTHS

Assigning the M_m outputs by the preceding graphical method will most likely not specify all of the outputs of the states of the SM. We shall now consider the problem of specifying the remaining outputs in order to minimize the length of the memory-span test of the output transformed table. Towards this end, the output pair-graph will be a useful device.

In forming the output pair-graph, branches are labeled by input-output pairs. A pair of states $S_i S_j$ is connected under input q to its successor pair $S_i^q S_j^q$ if $S_i^q \neq S_j^q$, if the outputs of S_i and S_j under q are the same, or one is not specified, or both are not specified. In other words, edges are not drawn if the specified outputs are different, or if both states have the same successor.

Following the above rules, we obtain the following output pair-graph for the SM of Figure 1.3d. The graph verifies that this machine has finite memory span, since there are no closed cycles.

The problem of specifying the outputs designated by \emptyset so as to minimize the memory-span test of the resulting flow table can, of course, be solved by trying all possible specifications of the don't-cares. Even though this approach would not be too tedious for this example, it would be prohibitive for larger tables.

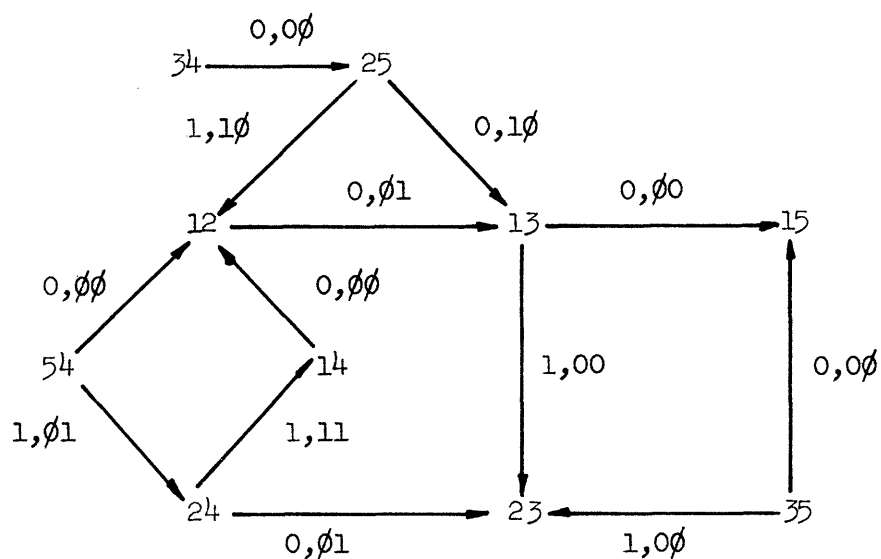


Figure 1.3(e) Output Pair-Graph for Fig. 1.3d

We shall now present a heuristic procedure which appears to lead to "short" realizations. First, obtain a lower bound on the length of the realization by finding the smallest integer $k \geq \log_{m+L} n$, where L is the level of the added output, m is the number of distinct inputs, and n is the number of states in the SM. Next, eliminate those complete chains on the graph which are no longer than k , since regardless of the specification of the don't-cares, at least one complete chain of length k or greater will remain. (A complete path is a directed path from a source node to a sink node.) For the problem at hand, $k \geq \log_4 5$, so that at least two register stages are required. Therefore, the chains $(35^0, 15)$ and $(35^1, 23)$ are eliminated; hence, node 35 is eliminated from the graph.

Next, consider only paths on the graph for which there are no don't-cares. In our case, these are $(13^1, 23)$ and $(24^1, 14)$. Now consider a node, say N_i , in one of the paths, say P_i . Let the length

of the chain from N_i to the end of P_i be L_i . Eliminate, except for P_i , all paths from N_i to a terminal pair which are no longer than L_i without destroying paths not involving N_i . These paths can be eliminated since the specified output chain is at least as long as the eliminated chain. For the graph of Figure 1.3e, we eliminate the links from 13 to 15 under input 0 and from 25 to 35 under 0.

Taking into account the links previously eliminated, the reduced graph of Figure 1.3f results.

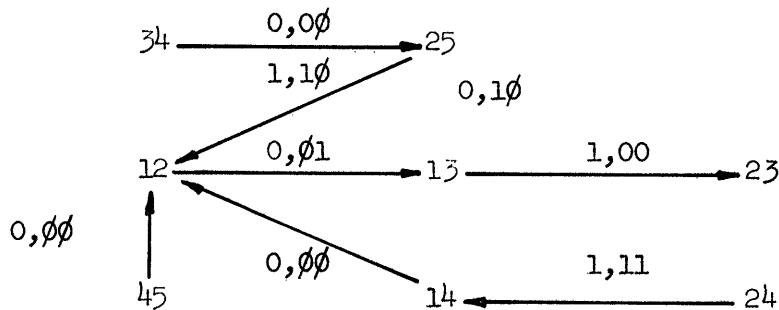


Figure 1.3(f) Reduced Graph

Next, find all state-input combinations for which a don't-care can be specified so that all remaining links of the state-input combination are eliminated. For example, specifying an output of 0 for state 5 under input 1 breaks all links from a pair containing 5 to any other remaining pair under input 1. Thus, under input 1, we let state 5 have an output of 0, and obtain the graph of Figure 1.3g.

All of the previous steps are in no way heuristic and can never lead to non-minimal solutions. However, at this point we resort to heuristic techniques: By specifying preferably only one output, attempt to cut the longest remaining chain in the graph in half. For the graph of Fig. 1.3g, the longest chain is $24^1, 14^0, 12^0, 13^1, 23$. We can cut

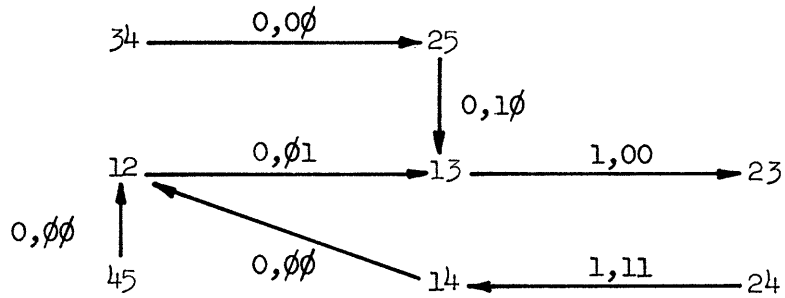


Figure 1.3(g) Graph Reduced Further

it in "half" by specifying the outputs of 1 and 4 under 0, i.e., break the link from 14 to 12, or by specifying the output of state 1 under 0, i.e., break the link from 12 to 13. Since we must specify only one output to break the link from 12 to 13, we choose to break it and specify the output for 1 under input 0 as 0. This results in the graph of Fig. 1.3h.

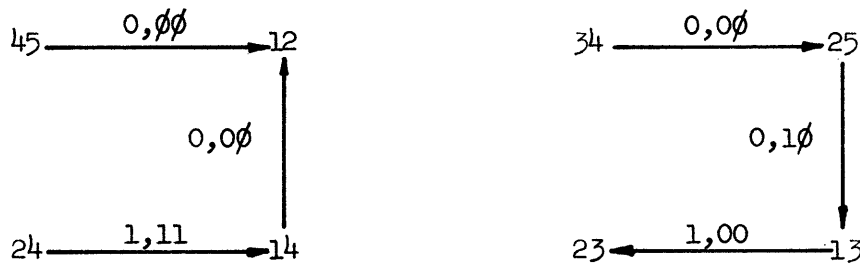


Figure 1.3(h) Graph Reduced Further

Finally, for this new graph, repeat the preceding four steps of eliminating all chains of length greater than k, etc., until all outputs are specified or until all chains greater than k are removed from the graph. Accordingly, link $(45^0, 12)$ is removed next. Then the output for state 4 under 0 is specified as 1, and the output of state 5 under 0 is specified as a 0. This results in the graph of Figure 1.3i.

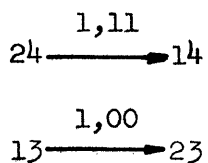


Figure 1.3i Result of Final Reduction

All outputs have now been specified, and the flow table shown in Figure 1.4 results.

	0	1
1	1/0	2/0
2	3/1	1/1
3	5/0	3/0
4	2/1	4/1
5	1/0	2/0

Figure 1.4 Final Flow Table

The length of the memory-span test for the output transformed table of Figure 1.5a is two, since the final graph of Figure 1.3i has no path longer than two, and no path longer than two has been eliminated in the process of finding that transformation. The memory-span test of the output transformed table is given in Figure 1.5b and the state assignment thus found is given in Figure 1.5c.

	00	01	10	11
1	1	-	2	-
2		3	-	1
3	5	-	3	-
4	-	2	-	4
5	1	-	2	-

Figure 1.5(a)

		00	01	10	11
12345		15	23	23	14
(0,0)	15	1	-	2	-
(0,1)	23	5	3	3	1
(1,0)	23	5	3	3	1
(1,1)	14	1	2	2	4

(b)

Input	Feedback		0	1
00	00	1	1	2
01	00	2	3''	1''
00	10	5	1	2
00	11	3	5	3'
01	10	3'	5'	3''
01	11	1'	1''	2''
10	00	5'	1	2
10	01	3''	5	3'
11	00	3''	5'	3''
11	01	1''	1''	2''
10	10	1''	1	2
10	11	2'	3	1'
11	10	2''	3''	1''
11	11	4	2'	4

(c)

Figure 1.5 Analysis of Machine of Figure 1.4

Since, by chance, the added output is input independent, the SM of Figure 1.3a can be realized with an input-fed, two-delay shift register in parallel with a two-delay FSR with input-independent feedback logic.

The following algorithm is a summary of the technique for specifying the don't-care outputs of a flow table so that the output transformed table has a "short" memory-span.

Algorithm 1.2:

1. Execute Algorithm 1.1 to determine the minimal level output which must be added so that the output transform has finite memory. Choose that set of adjacency maps from among those which correspond to the minimum level realization that specifies the fewest outputs in the flow table. Form the input-output pair-graph by connecting node $S_i S_j$ to its successor node $S_i^1 S_j^1$ under input I^1 iff $S_i^1 \neq S_j^1$ and the outputs for S_i and S_j under I^1 are the same, one is not specified, or both are not specified.
2. Let k be the smallest integer such that $k \geq \log_{m+L}(n)$, where L is the level of the added output, m is the number of inputs, and n is the number of states in the SM. Eliminate all complete paths of length no greater than k . (A complete path is one which starts at a tip (source) and ends at a terminal (sink) of the graph.)
3. Find all paths having completely specified outputs. If any node N_i in this path has another path P_i leading to a terminal node of length shorter than the length of the path from N_i to the end of the specified path, eliminate P_i .
4. If all links of a given state/input combination can be broken by specifying the output of that state under that input, then specify it to break these links. Remove all links caused by specifying the output and return to Step 2. If no output can be specified according to this rule go to Step 5.
5. Try to cut the longest remaining chain in the graph in "half" by specifying as few outputs as possible. If there is more than one chain, then choose that chain which can be cut most

nearly in half by specifying the fewest outputs. Break those links caused by specifying the outputs. Go to Step 2 if the remaining graph still has don't-cares on it.

6. The length of the realization will be the length of the longest chain in the remaining graph or k , whichever is larger.

1.5 A HEURISTIC TECHNIQUE

As can easily be seen, the techniques of Algorithms 1.1 and 1.2 for finding a realization, though systematic, require a discouraging amount of work. The first part of the algorithm requires generating a graph with $(n^2-n)/2$ nodes, listing all cycles in this graph, and then forming the "logical product" of the propositions which when true break all cycles. Hence, a 10-state SM leads to a 45-node graph. A 45-node graph can have approximately 7.4×10^{54} cycles. Thus, the first part of the algorithm, though mathematically complete, leaves a bit to be desired computationally.

To avoid using Algorithm 1.1, we have developed a heuristic technique which appears to yield "good" solutions for the minimum level output. In this algorithm it will be convenient to use the term "output-closed-loop," which is a closed loop in the output pair-graph with completely specified outputs and at least one pair of identical outputs.

Algorithm 1.3:

1. Form an input-output pair-graph with all outputs initially specified as don't-cares. Later, when an output is added, delete a link which, due to this specification, has different outputs on it.

2. If each of k' states maps to itself under any particular input, then at least a k' -valued output is required so that the output transform has finite memory. Assign different outputs to the "self-loops" in a column, i.e., to those next-state entries which equal the row label. If a column does not contain a self-loop, then assign an output of 0 to state 1 in that column.
3. Specify the output of S_i under input I_j , if it can be specified without increasing the output level, so that all cycles which contain that state-input pair are eliminated.
4. On the input-output pair-graph, eliminate all nodes not in cycles, and then find a short cycle. (We refrain from saying the shortest cycle since this constraint leads to computational difficulty.) If possible, choose a node which has one output already specified in that cycle and specify the other output so that the link is broken; however, do not specify it so that an output-closed-loop is formed on the resulting graph. If none of the present levels suffice, either add another level or try another node. Go to Step 3 and repeat this process until all loops are broken on the output pair-graph; then use Algorithm 1.2 to specify the other outputs.

This algorithm is also computationally quite complex. However, we are dealing with a complex problem and unfortunately must expect this.

As an example of this technique, consider again the SM of Figure 1.3a. First, we note that states 3 and 4 form self-loops, i.e., each

maps to itself, under input 1 and that state 1 forms a self-loop under input 0. In Step 2, we assign an output of 0 to 3 and a 1 to 4 under input 1, and assign an output 0 to state 1 under input 0. Doing this leads to the output pair-graph of Figure 1.6a, where, in accordance with step 4, links and nodes not in loops are not drawn. The cycle 12

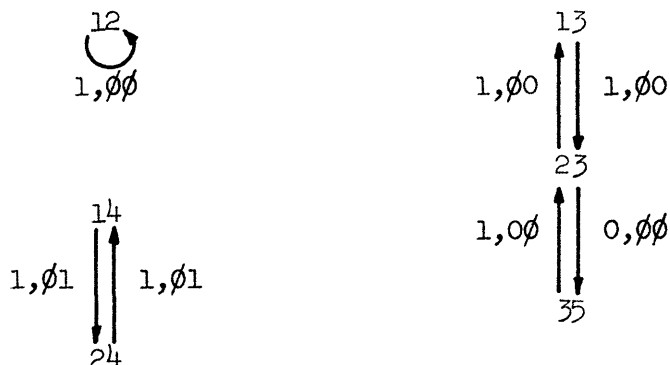


Figure 1.6(a) Partial Output Pair-Graph

under input 1 is the shortest cycle of the graph; hence, under input 1, let 2 map to 1 with an output of 0 and 1 to 2 with an output of 1. The flow table then becomes that shown in Figure 1.6b and the graph is

	0	1
1	1 0	2, 1
2	3 \emptyset	1, 0
3	5 \emptyset	3, 0
4	2 \emptyset	4, 1
5	1 \emptyset	2, \emptyset

Figure 1.6(b) Partial Output Specification

simplified to that of Figure 1.6(c). Hence, let 5 map to 2 under input

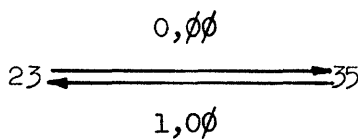


Figure 1.6(c)

1 with an output of 1. The flow table becomes

	0	1
1	1 0	2,1
2	3 \emptyset	1,0
3	5 \emptyset	3,0
4	2 \emptyset	4,1
5	1 \emptyset	2,1

Figure 1.6(d)

The don't-cares are now specified according to Algorithm 1.2. Doing this leads to the flow table of Figure 1.63. The output transform of this flow table has length-two memory-span.

	0	1
1	1 0	2 1
2	3 1	1 0
3	5 1	3 0
4	2 0	4 1
5	1 0	2 1

Figure 1.6(e)

Let us consider, as a final example of these techniques, the flow table of Figure 1.7. Since there are eight self-loops in column 0, we know that an 8-level output must be added to make this flow table have a finite-memory output transform. Thus, one level per state is needed, which corresponds to an arbitrary realization containing three binary delays. Hence, the best FSR realization this technique finds is not a FSR realization at all. Our technique does so poorly in this example because it uses only one of the forms of state splitting:

	0	1
1	1	2
2	2	3
3	3	4
4	4	5
5	5	6
6	6	7
7	7	8
8	8	8

Figure 1.7 A SM

splitting during the memory-span test. If one allows state splitting before the memory-span test, then from Friedman's work we know that any SM can be realized with two shift-registers, one which stores inputs, and a binary FSR. But, as mentioned before, these realizations can be prohibitively long.

CHAPTER 2

FEEDBACK SHIFT-REGISTERS AND THE DIAGNOSIS OF SEQUENTIAL MACHINES

After synthesizing a sequential machine (SM), the designer must determine whether the circuit performs the desired input-output transformation. Furthermore, it is often necessary to check a sequential circuit periodically for malfunctions. Before the advent of integrated circuits, this checking, or diagnosing, was frequently facilitated by attaching probes to the circuit itself or by cutting some of the feedback loops in the circuit. However, neither of these techniques can be easily implemented when the SM is realized with an integrated circuit. One is thus forced to develop new techniques for diagnosing SM's which are synthesized with integrated circuits. One fruitful approach has been to modify the flow table by adding extra inputs (Ref. 4) or outputs (Ref. 3) so that the modified flow table is relatively easy to check. Using the classical switching theory standards of minimizing the number of components in a circuit, this approach would have questionable merit. However, as is well known, the classical standards of synthesis do not apply to integrated circuits.

It seems reasonable that the next logical step to be taken in flow-table modification theory is to investigate the possible benefits incurred by state splitting. At first, one might suspect that increasing the number of states in a flow table would naturally lead to circuits which are harder to diagnose. However, in Section 2.1 we shall show that this need not be the case by proving that feedback-

shift-register (FSR) realizations are relatively simple to diagnose if one modifies the circuit only trivially. Hence, if by splitting states one can realize the flow table with a FSR, then the circuit will usually be easier to check.

Our discussions concerning diagnosis of SM's will not be limited to FSR realizations. After a comment on flow-table modification by the addition of one input, we shall investigate modifying a flow table by splitting states and by adding a single binary output so that it has "good" diagnosing properties.

Although we shall repeat the definition of the basic terms used concerning SM diagnosis, we shall not, for the sake of brevity, present a survey of the general theory. Our definitions are taken from Ref. 3, but our approach will differ from Kohavi and LaVallee's in that we shall split states as well as add outputs.

An experiment on a SM is the application of an input sequence to a SM and the recording of the corresponding response at its output terminals. An experiment which allows one to determine whether or not a SM is operating correctly is said to be a checking experiment.

Hennie (Ref. 5) has shown that SM's which have distinguishing sequences have relatively "short" checking experiments. (A distinguishing sequence applied to an SM generates a different output sequence for each possible starting state of the SM. Hence, if the present state of the SM is unknown, by applying the distinguishing sequence and observing the corresponding output sequence, one can determine that state, if the SM is working properly.) For example, consider the machine of Figure 2.1a. The input sequence 11 is a distinguishing sequence,

	0	1
A	A/0	B/0
B	A/0	C/1
C	A/0	C/0

Figure 2.1(a) A SM which Has a Distinguishing Sequence

because, when applied to starting state A, it results in the output sequence 01; to B, 10; and to C, 00. Unfortunately, not all SM's have distinguishing sequences. For example, the SM of Figure 2.1b has no distinguishing sequence.

	0	1
A	A/0	B/0
B	A/0	C/0
C	A/0	D/0
D	A/1	D/0

Figure 2.1(b) A SM which Does Not Have a Distinguishing Sequence

To overcome the problem of long checking sequences for SM's which do not have distinguishing sequences, Kohavi and LaVallee modify the flow table by adding additional outputs so that the modified flow table has distinguishing sequences. Since one could let all the state variables be additional outputs, this can always be done. In their paper, Kohavi and LaVallee suggest that SM's for which all sequences of some length p are distinguishing sequences have "good" checking experiments. Kohavi and LaVallee designate a SM definitely diagnosable if any sequence of length p is a distinguishing sequence. Furthermore, they introduce techniques for adding a "small" number of outputs to a flow table to make it definitely diagnosable.

2.1 DIAGNOSING FSR REALIZATIONS

If a SM is realized with an L-level, k-delay FSR, then the flow table corresponding to the realization can be made definitely diagnosable by adding a state-determined (input independent), L-valued output. The technique is simple: In the realization flow table, the output digit Y_k is added to all the entries in the present state row with assignment $Y_k Y_{k-1} \dots Y_1$. This output addition corresponds physically to making the output of the last delay of the shift register be an additional output of the SM. (See Figure 2.2.) Since applying any k-digit input sequence to the SM will cause the assignment of the starting state to be serially shifted out of the register, the modified flow table is definitely diagnosable.

As an example of this output addition, consider the SM of Figure 2.1b. It has the FSR realization flow table of Figure 2.3. Only the

		0	1
000	A	A/0	B/0
001	B	A ¹ /0	C/0
010	A ¹	A ² /0	B ¹ /0
011	C	A ³ /0	D/0
100	A ²	A/1	B/1
101	B ¹	A ¹ /1	C/1
110	A ³	A ² /1	B ¹ /1
111	D	A ³ /1	D/1

Figure 2.3 A Modified SM

added output symbol used for diagnosing is shown and the original output from the SM of Figure 2.1b is omitted. Hence, if the output sequence $Z(t), Z(t+1), Z(t+2)$ is 0,1,1, we conclude that the SM was in

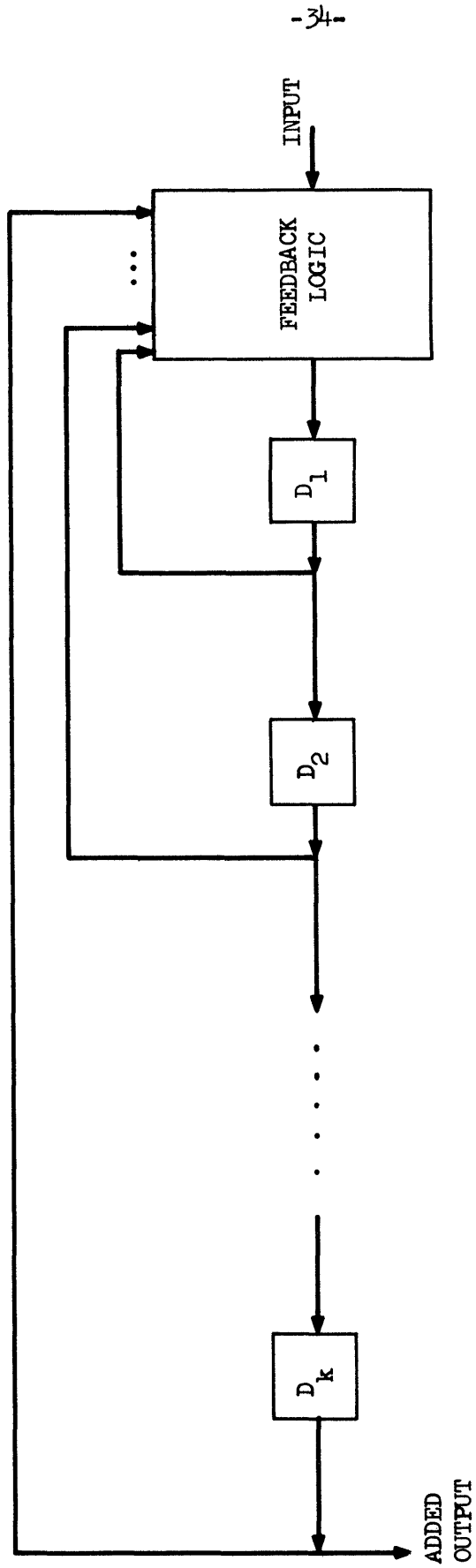


Fig. 2.2 FSR with Additional Output

state C at time t. It is interesting to note that using Kohavi and LaVallee's technique, if states are not split, two binary outputs must be added to the flow table of Figure 2.1b to make it definitely diagnosable.

Kohavi and LaVallee prove, by example, that taking a SM through all its transitions does not, in general, constitute a checking sequence. However, in a FSR realization modified as described above, if one assumes that the malfunction in the shift register part of the realization corresponds to a constant input to a delay, i.e., connection wires between adjacent delays opening or shorting, then it is easy to see that taking the SM through all its transitions - now all the transitions for all the equivalent states - and then applying any k inputs to check the final state does constitute a checking sequence. This is the case since, under these assumptions, we need only check for the following types of errors: 1. a shorted input to a delay, 2. an opened input to a delay, 3. malfunctions in the feedback logic. The first two types of errors will be easily detected as the added output would soon take on only all 0's or all 1's. Malfunctions in the feedback logic will be observed from the added output by taking the SM through all its transitions and then applying any k inputs. However, if the checking sequence must check for any possible malfunction of the circuit, then taking the SM through all its transitions and then applying k inputs does not constitute a checking sequence. The flow tables and the trial checking sequence of Figure 2.4 prove this. However, this sequence would be a checking sequence for the SM of Figure 2.4a under the previous assumptions.

		0	1
00	1	1/0	2/0
01	2	3/0	4/0
10	3	1/1	2/1
11	4	3/1	4/1

(a) Machine A

		0	1
1	1	1/0	2/0
2	2	3/0	4/0
3	3	4/1	2/1
4	4	1/1	3/1

(b) Machine B

Figure 2.4 Two SM's

Inputs to Machines A and B	0	1	0	1	1	1	0	0	0	0	0	
State of Machine A	1	1	2	3	2	4	4	3	1	1	1	1
State of Machine B	1	1	2	3	2	4	3	4	1	1	1	1
Outputs of Machines A and B	0	0	0	1	0	1	1	1	0	0	0	0

In summary, we have shown:

Lemma 2.1: A SM realized with a binary FSR can be made definitely diagnosable with the addition of at most one binary output to the realization flow table. This output is taken directly from the output of the last delay of the shift register and requires no additional logic.

Obviously, Lemma 2.1 can be extended to parallel, multi-level FSR realizations.

Lemma 2.2: If a SM is realized with j FSR's, then the flow table corresponding to the FSR circuit can be made definitely diagnosable by adding at most j , state-dependent outputs to each state.

It is important to note that one makes the flow table corresponding to the FSR realization definitely diagnosable by the addition of these outputs. This means that one could make a flow table definitely

diagnosable by adding the last bit of the FSR state assignment as a new output, realize the flow table in any arbitrary fashion, but still use the added output for diagnosing purposes. From previous discussions, we must expect that the flow table corresponding to the FSR realization will most likely have many equivalent states. However, as we have seen from the example of Figures 2.1 and 2.3, expanding the flow table might reduce the number of additional outputs required to make the SM definitely diagnosable. An interesting example of the potential of this reduction and the attendant explosion in the number of states is shown in Figure 2.5. Using Kohavi and LaVallee's technique, we would find

	0	1
1	1/0	2/0
2	1/0	3/0
3	1/0	4/0
.	.	.
.	.	.
n-1	1/0	n/0
n	1/0	n/1

Figure 2.5 Another SM

that $\log_2 n$ additional binary outputs are required to make this SM definitely diagnosable. However, since this flow table is binary FSR realizable with a FSR of length $n-1$, we find that if one allows state splitting, only one additional binary output is required. However, the resulting realization flow table has 2^{n-1} states.

Using our knowledge concerning the feedback index of SM's, we can establish an upper bound on the number of outputs one must add to make and SM definitely diagnosable. From Friedman's work (Ref. 2) we know

that any L-column SM can be realized with an input-fed L-level shift register in parallel with a binary FSR. Hence, we can generate a FSR realization flow table for any L-input SM which can be made definitely diagnosable with the addition of at most one 2L-valued output. Hence, we have:

Lemma 2.3: If one allows state splitting, any L-input SM can be made definitely diagnosable with the addition of at most one 2L-valued output.

As yet, however, no SM has been found which requires the addition of 2L output values to make it definitely diagnosable.

2.2 FLOW-TABLE MODIFICATION BY COLUMN ADDITION

C. R. Kime (Ref. 4) introduced a second approach to the diagnosing problem of SM's. Kime does not attempt to modify a SM so that it is definitely diagnosable, but only so that it has a distinguishing sequence. One of his modifications is based on the following.

We shall say that SM_1 column contains SM_2 if deleting some of SM_1 's columns creates SM_2 and if SM_2 has no equivalent states. Kime showed that if SM_1 column contains SM_2 and if no two states in SM_2 have the same successor/output pair under any of its inputs, then both SM_2 and SM_1 have a distinguishing sequence.

For an example of the use of Kime's Theorem, consider the SM of Figure 2.6a. Deleting columns 0 and 2 from the SM of Figure 2.6a creates the SM of Figure 2.6b which is reduced and has a distinguishing sequence since no two of its states have the same successor/output pair. Note that even though this single-column SM is not strongly connected, the

	0	1	2
A	A/0	B/0	B/0
B	B/0	C/1	C/0
C	A/0	D/1	B/0
D	D/1	D/0	A/1

Figure 2.6(a) Another SM

	1
A	B/0
B	C/1
C	D/1
D	D/0

Figure 2.6(b) A SM Column Contained by the SM of Figure 2.6a

original SM is; thus, the distinguishing sequence and the technique which Hennie introduced for diagnosing strongly connected SM's which have distinguishing sequences can be used. Using this result, Kime then suggested appending to any strongly connected SM a single-column, irreducible SM which has a distinguishing sequence. In adding this column, Kime assumes that the SM has a complete, k-bit state assignment and thus has 2^k states. The column Kime adds is the shift register, divide-by-two column. In other words, state S_i , with binary assignment b_i , maps to the state with assignment $\left\lfloor \frac{b_i}{2} \right\rfloor$, ($\lfloor \rfloor$ signifies integer part of). The output for state S_i is the rightmost digit of its state assignment (e.g., for the assignment 01, the output would be 1). For an example of this procedure, consider the SM of Figure 2.7a. The divide-by-two column is added, resulting in the SM shown in Figure 2.7b.

Kime's divide-by-two simply shifts the state assignment one digit to the right and introduces a zero as the new leftmost digit. Thus, the

		0	1
00	A	A/0	C/0
01	B	A/0	D/1
10	C	B/1	A/1
11	D	B/1	C/0

Figure 2.7(a) A SM

		0	1	2
00	A	A/0	C/0	A/0
01	B	A/0	D/1	A/1
10	C	B/1	A/1	B/0
11	D	B/1	C/0	B/1

Figure 2.7(b) The SM of Figure 2.7a with a Column Modification column has a distinguishing sequence of length k that generates an output sequence which is the state assignment of the initial state.

We suggest here a minor modification of Kime's technique. We propose adding to the SM a column which is a single cycle of length 2^k with outputs (added) so that any sequence of k inputs of this added column will generate an output sequence which is the state assignment of the initial state. Furthermore, since this column is a cycle column, its operation can be checked by a sequence of 2^k+k consecutive inputs of that column. Kime's divide-by-two column is not as simple to check. However, the main advantage of our suggestion is that adding it to any SM makes it strongly connected. Hence, the rather unfortunate constraint of strong connectedness usually assumed in diagnosing techniques can be discarded.

We shall now show that there exists a 2^k -length cycle column such that applying k inputs to any state S_i of that column will result in an

output sequence which is the state assignment of S_i . This follows from the well established fact that a k -delay, binary FSR can generate a cycle of length 2^k . Hence, the added column is one of the 2^k -length, FSR cycles. The output for the successor of a state in the column is the bit of its assignment which is shifted out of the register. It should be noted that if a SM is FSR realizable, these columns can be added without destroying the FSR realizability of the original SM.

2.3 FLOW-TABLE MODIFICATION BY OUTPUT ADDITION

Kime's technique of adding an input to a SM proves that any SM can be augmented so that it has a distinguishing sequence with the addition of at most one line to the SM - an input line. We shall now show that by splitting states, any SM can be modified so that it has a distinguishing sequence with the addition of at most one binary output line. Even though we shall use FSR techniques to generate the modified flow table, it will, in general, not be FSR realizable.

From Lemma 2.1, we know that if a SM is FSR realizable, then the realization flow table can be made definitely diagnosable by assigning as a new output digit for each state in the flow table that bit of its assignment which is shifted out of the register. Recall that even if the modified flow table were not realized with a FSR, it is still definitely diagnosable.

For an example of the use of Lemma 2.1, consider the flow table of Figure 2.8a. This flow table is binary FSR realizable if one splits states; hence, by splitting states it can be made definitely diagnosable with the addition of one binary output. The realization flow table with the added output is shown in Figure 2.8b. Now, consider the flow table

		0	1
1		1	2
2		1	3
3		1	3

Figure 2.8(a) Another SM

			0	1
00	1		1/0	2/0
10	1 ¹		1/1	2/1
01	2		1 ¹ /0	3/0
11	3		1 ¹ /1	3/1

Figure 2.8(b) FSR Realization of the SM of Figure 2.8a with Outputs Added

of Figure 2.9a. Even though this flow table is not binary FSR realizable,

		0	1	2
1		1	2	3
2		1	3	3
3		1	3	2

Figure 2.9(a) Another SM

it contains, by deleting column 2, a binary FSR realizable flow table.

Hence, by splitting states and adding the outputs as shown in Figure 2.9b, the flow table of Figure 2.9a can be made to have distinguishing sequences using inputs 0 and 1.

			0	1	2
1			1/0	2/0	3/
1 ¹			1/1	2/1	3/
2			1 ¹ /0	3/0	3/
3			1 ¹ /1	3/1	2/

Figure 2.9(b) An Output Modified Flow Table

In general, by splitting states and adding a single binary output, a SM can be modified to have a distinguishing sequence if deleting some of its columns generates a binary FSR realizable SM. But, from Haring (Ref. 6) we know that every single-column SM is binary FSR realizable. Hence, if one deletes all but one column, say column 0, of a SM, then finds the FSR realization of the resulting single-column SM, adds the appropriate output to each state, and then appends the old columns to the single-column FSR realization, the original machine will have an all-zero distinguishing sequence.

For an example of this technique, consider the flow table of Figure 2.10a. Even though the flow table of Figure 2.10a is not binary

	0	1
1	1	5
2	3	1
3	4	3
4	2	7
5	1	2
6	6	2
7	7	6

Figure 2.10(a) Another SM

FSR realizable, the SM formed by deleting column 1 is. Its realization flow table, with distinguishing outputs added, is shown in Figure 2.10b. Appending the original column 1 to the above FSR realization, we find the SM shown in Figure 2.10c. The sequence 0000 is a distinguishing sequence for this SM.

		0
0000	1	1/0
1101	2	3/1
1011	3	4/1
0110	4	2/0
1000	5	1/1
1010	6'	6/1
0101	6	6'/0
1001	7'	7/1
0010	7''	7'/0
0100	7	7''/0

Figure 2.19(b) The Binary FSR Realization of Column Zero of the SM of Figure 2.10a

		0	1
1	1/0	5	
2	3/1	1	
3	4/1	3	
4	2/0	7, 7', or 7''	
5	1/1	2	
6	6'/1	2	
6'	6/0	2	
7	7'/1	6 or 6'	
7'	7''/0	6 or 6'	
7''	7/0	6 or 6'	

Figure 2.10(c) The SM with Output Added so that It Has the All-Zero Distinguishing Sequence

Hence, considering all the preceding arguments we have proved the following theorem.

Theorem 2.1: By splitting states, any SM can be made to have a distinguishing sequence consisting of a single repeated symbol with the addition of a single binary output.

It is interesting to note that the increase in the number of delays necessary to realize the SM modified to have the all-zero distinguishing sequence is not too great. It can be shown that this increase is no greater than $\log_2 \log_2(n)+1$, where n is the number of states in the original SM. Also, a SM can be exhibited which requires the addition of at least $\log_2 \log_2(n)-1$ delays if it is modified to have the one-of-a-kind distinguishing sequence.

2.4 SUMMARY

In this chapter, we have suggested various ways to modify a sequential machine (SM) so that its operation is easy to check. First, we showed that feedback shift register (FSR) realizations are relatively easy to diagnose if one makes the bit which is shifted out of the shift register an additional output of the circuit.

Next, we investigated ways of modifying flow tables so that they have distinguishing sequences. We first suggested a slight modification of Kime's procedure of adding an extra input to the SM. Then, we showed that by splitting states and adding a single binary state-determined output, any SM can be modified to have a distinguishing sequence consisting of a single repeated symbol. Since one desires to minimize the number of input and output lines in an integrated circuit, this result is quite interesting.

Perhaps, however, the most rewarding aspect of this chapter has been the ease with which "non-FSR" problems were solved by using FSR techniques.

REFERENCES

1. Susskind, A. K., et. al., "Threshold Elements and the Design of Sequential Switching Networks," Technical Report No. RADC-TR-67-255, July 1967, Section 3.4.
2. Friedman, A. D., "Feedback in Synchronous Sequential Switching Circuits," IEEE Trans. on Electronic Computers, June 1966, Vol. EC-15, No. 3, pp. 354-368.
3. Kohavi, Z. and P. LaVallee, "Design of Diagnosable Sequential Machines," AFIPS Conference Proceedings, April 1967, Vol. 30, pp. 713-718.
4. Kime, C. R., A Failure Detection Method of Sequential Circuits, Department of Electrical Engineering University of Iowa Technical Report 66-13, January 1966.
5. Hennie, F. C., "Fault Detecting Experiments for Sequential Machines," Proceedings of the Fifth Annual Symposium on Switching Theory and Logical Design, 1964, pp. 95-110.
6. Haring, D. R., Sequential-Circuit Synthesis: State Assignment Aspects, M.I.T. Press, Cambridge, Mass., 1966.

CHAPTER 3

BI-DIRECTIONAL FEEDBACK SHIFT-REGISTER SYNTHESIS OF SEQUENTIAL MACHINES

In this chapter, we give a new technique for realizing a sequen-
tial machine (SM) in the form of a bi-directional feedback shift
register (BFSR's), under particular constraints to be discussed below. A BFSR is a serial connection of k unit delays labelled 1 through k from right to left and interconnected so that at the occurrence of a left-shift signal the contents of the i^{th} delay is moved into the $(i+1)^{\text{th}}$ delay and at the occurrence of a right-shift signal the contents of the i^{th} delay is moved into the $(i-1)^{\text{th}}$ delay. If a SM is realized with a BFSR, then the input to the first delay in the case of left shifting and last delay in the case of right shifting is generated by combinational logic which has access to the contents of the delays and the inputs to the SM. We shall insist that the register shift after each input occurrence. We shall be concerned not only with binary FSR realizations, but also with r -level realizations in which the input to the first delay can be any one of r values, represented as $0, 1, 2, \dots, r-1$.

Ideally, the techniques for synthesizing SM's with BFSR's should be general enough to include arbitrary freedom in choosing shift direction, i.e., as a function of both the state assignment and input, and should include the possibility of assigning more than one code per state, i.e., the introduction of equivalent states. Though techniques

have been developed which gave insight into the equivalent state problem associated with the unidirectional FSR synthesis of SM's (Ref. 1), we have not been able to extend them to apply to BFSR synthesis. Hence, except for certain proofs concerned with the lack of realizability of certain SM's with BFSR's, we shall be concerned here only with BFSR synthesis techniques in which each state is assigned exactly one code.

Moreover, we shall restrict our attention to synthesis techniques in which the shift-direction is input determined. For example, under input 0 the shift direction is always left and under input 1 the shift direction is always right. We impose this restriction for computational ease and because the form is relatively simple to realize, since there need be no feedback logic to determine the shift direction. The case in which the shift direction is state determined has been previously investigated by us (see Ref. 2) and will not be repeated here.

First, we will develop an algorithm for the synthesis of SM's with only two input symbols. Then, without loss of generality, we can assume that the register shifts left under input 0 and right under 1. We shall designate such a column contingent BFSR realization as CBFSR. (The case of shifting right, say, under both inputs has also been previously investigated in Ref. 2 and will not be presented here.) The SM of Figure 3.1 can be realized in the form considered here and the resulting state assignment is shown.

$y_3 y_2 y_1$		0	1
000	1	2	5
001	2	3	1
010	3	6	6
011	4	8	6
100	5	1	7
101	6	3	7
110	7	6	8
111	8	7	4

Figure 3.1 An Example of a BFSR Realization

3.1 THE SYNTHESIS ALGORITHM

In summary, our goal is to develop an algorithm with which one can determine whether a SM is CBFSR realizable. If the SM is CBFSR realizable, this algorithm should indicate how to generate the appropriate state assignments (SA's). Under the restriction of one assignment per state, one finds that partition theory, developed by Hartmanis and Stearns (Ref. 3), is very convenient in determining whether a CBFSR realizable SA exists. A block of partition P_i on the set of the states of a SM has in it all those states which are coded the same in state variable Y_i . For example, for the SM of Figure 3.1, $P_1 = (1357, 2468)$.

Towards finding the SA, or more appropriately the state variable partitions, we define the operator $J^{x\bar{x}}$, where x equals 0 or 1 and \bar{x} equals, respectively, 1 or 0. The operator $J^{01}(J^{10})$ acts on the partition to its left by joining each state in a block with its 01-(10) successor. (The 01-successor of a state is the state that it maps to

under the input sequence 01. E.g., the 01 successor of state 2 of the SM of Figure 3.1 is state 6.) All blocks thus formed which have common members are joined. For example, if $P = (12, 34, 56, 78)$, then $PJ^{01} = (1256, 3478)$.

The following lemma follows immediately from the definition of the operator $J^{\bar{x}}$ and from the shift constraint of the CBFSR.

Lemma 3.1: If a SM is CBFSR realizable and if R is such that $R \leq \prod_{j=a}^b P_j$, then $RJ^{01(10)} \leq \prod_{j=a}^b P_j$ if $a \neq 1$ ($b \neq k$) or $RJ^{01(10)} \leq \prod_{j=a}^{k-1} P_j$ if $b = k$ ($\prod_{j=2}^b P_j$ if $a = 1$).

Lemma 3.1 can be used to start to specify the state-assignment partitions. For example, for the SM of Figure 3.1, we know that the zero (0) partition (1,2,3,4,5,6,7,8) satisfies the following relation

$$(0) = \prod_{i=1}^3 P_i$$

and hence that

$$(0)J^{01} = (1,26,37,5,4,8) \leq \prod_{i=1}^2 P_i$$

$$(0)J^{10} = (1,2,34,56,7,8) \leq \prod_{i=2}^3 P_i$$

and by the union of partitions that

$$(0)J^{01} \cup (0)J^{10} = (1,265,347,8) \leq P_2$$

Another operator useful in FSR synthesis is T^x . The operator T^x acts on the partition to its left by replacing each state in each block

by its successor under input x , joining blocks thus formed with common members, and adding states inaccessible under input x as singletons. The following lemma follows directly from the definition of the operator T^x and the shift constraints of a CBFSR realization.

Lemma 3.2: If a SM is CBFSR realizable and $R \leq \prod_{j=a}^b P_j$, then $RT^0 \leq \prod_{j=a+1}^{b+1} P_j$ and $RT^1 \leq \prod_{j=a-1}^{b-1} P_j$, where $b+1$ is set equal to k if b equals k and $a-1$ is set equal to 1 if a equals 1 .

Similarly, the operator T^{-x} acts on the partition to its right by replacing each state in each block by all of its predecessors under input x and joining blocks thus formed with common members. States which have no successors, i.e., a state which maps to a don't-care under input x , are added as singletons in $P_i T^{-x}$. Lemma 3.3 follows directly from the definition of the operator T^{-x} and the shift constraint in a CBFSR realization.

Lemma 3.3: If a SM is CBFSR and if $R \leq \prod_{j=a}^b P_j$, then $RT^{-0} \leq \prod_{j=a-1}^{b-1} P_j$ and $RT^{-1} \leq \prod_{j=a+1}^{b+1} P_j$, where $b+1$ is set equal to k if b equals k and $a-1$ is set equal to one if a equals 1 .

Lemmas 3.2 and 3.3 can be used to obtain further information concerning the state-assignment partitions. For example, for the SM of Figure 3.1 we previously found that $P_2 \geq (1,256,397,8) = R$.

Therefore from Lemmas 3.2 and 3.3

$$RT^1 \cup RT^{-0} = (5,173,268,4) \leq P_1$$

$$RT^0 \cup RT^{-1} = (2,134,568,7) \leq P_3$$

As can easily be seen, the partitions thus found constitute a four-level CBFSR realization. If a binary realization is desired, then blocks of partitions must be joined. The only joinings so that a binary SA is found leads to the one shown in Figure 3.1.

However, in using the operators as shown above, one might lose some information. To avoid this, we now wish to define the union of operators which will be done so that $P(T^{-y} U T^x) \geq PT^{-y} U PT^x$, where x and y are 1 or 0. The operator $T^{-y} U T^x$ acts on the partition to its left by replacing each block by both of its successors under x and all of its predecessors under y , joining blocks thus formed with common members, and adding states which are both inaccessible under x and have a don't-care successor under y as singletons. From the definition of the operators it follows that

Lemma 4.4: If a SM is CBFSR realizable and $R \leq \prod_{j=a}^b P_j$, then

$$R(T^{-0} U T^1) \leq \prod_{j=a-1}^{b-1} P_j$$

$$R(T^{-1} U T^0) \leq \prod_{j=a+1}^{b+1} P_j$$

where $b+1$ is set equal to k if b equals k and $a-1$ is set equal to 1 if a equals 1.

The operators $J^{\overline{xx}}$ and T^x allow one to obtain information relevant to the SA partitions. The only question remaining to be answered is how to use them effectively. Algorithm 3.1 presents an organization for their use that has proved to be quite efficient in finding the state-assignment partitions for CBFSR realizable SM's. The strategy of

the algorithm is to first find bounds on the middle SA partition $P_{\lfloor \frac{k}{2} \rfloor}$, and then to use this partition to obtain bounds on the remaining state-assignment partitions. In the algorithm the notation S_i^j will refer to a partition which is guaranteed to be contained by the partition $\prod_{r=i}^j P_r$.

Algorithm 3.1:

1. Let $K = \lceil \log_j n \rceil$, where n is the number of states in the SM and j is the desired level of realization.
2. Form $S_2^{k-1} = (0) [J^{01} \cup J^{10}] \cup (0) T^{-1} \cup (0) T^{-0}$
3. Starting with S_2^{k-1} , form $S_{i+1}^{k-i} = S_i^{k-i+1} [T^0 \cup T^{-1}] \cup S_i^{k-i+1} [T^1 \cup T^{-0}]$ until either $i+1 = k-i$, if k is odd, or until $i+2 = k-i$ if k is even. In the latter case, form $S_{\frac{k}{2}}^{\frac{k}{2}} = S_{\frac{k}{2}+1}^{\frac{k}{2}+1} \cup S_{\frac{k}{2}}^{\frac{k}{2}+1} [T^1 \cup T^{-0}]$. (If k is odd, then the partition found in Step 3 will be a lower bound on $P_{\frac{k+1}{2}}$; if k is even, then the partition found will be a lower bound on $P_{\frac{k}{2}}$.)
4. From $S_{\frac{k+1}{2}}^{\frac{k+1}{2}}$ (or $S_{\frac{k}{2}}^{\frac{k}{2}}$) find a bound on the rest of the P_i by using the operators $(T^{-1} \cup T^0)$ and $(T^{-0} \cup T^1)$. For example,

$$S_{\frac{k+1}{2}}^{\frac{k+1}{2}} (T^{-1} \cup T^0)^j = S_{\frac{k+1}{2}+j}^{\frac{k+1}{2}+j}.$$
5. Given $S_1^1 = P$ and $S_k^k = Q$, form $S_1^1 (T^0 \cup T^{-1})^{k-1} = R$ and $R(T^1 \cup T^{-0})^{k-1} = M$. If $M = P$ and $R = Q$, go to Step 6 and let

$S_j^j = R(T^1 U T^{-0})^{k-j}$. If not, let $S_k^k = R$ and $S_1^1 = R(T^1 U T^{-0})^{k-1}$ and return to the beginning of Step 5.

6. The S_i thus found represent bounds on the SA partitions resulting from the shifting constraints of the CBFSR form.
7. If $\prod_{i=1}^k S_i = (0)$, then a valid state-assignment has been found with as many levels as there are blocks in that S_i having the largest number of blocks.
8. If a realization with fewer levels is desired, then attempt to join blocks of the S_i in a manner compatible with $T^x T^{-x}$ and J^{xx} to form new partitions so that the intersection of the resulting partitions is zero.
9. If an L-level, but not (L-1)-level, realization can be thus found and an (L-1)-level realization is desired, then increase k by one and return to Step 2 and attempt to find a lower level realization.

Since all of the operators have been previously demonstrated, we shall not bother to present an example of the use of Algorithm 3.1.

3.2 EQUIVALENT STATES

Until now we have been considering only realizations of SM's with BFSR's in which each state has exactly one assignment. As was shown for the unidirectional case (Ref. 1), if one allows more than one assignment per state, then many more SM's can be realized with BFSR's.

Given that one can have arbitrarily many codes per state and has complete freedom in shift direction, i.e., the shift direction can be a function of the state assignment and the input, then it is not yet known if there exists a two-column SM which is not BFSR realizable. If, however, one restricts the shifting as in the CBFSR case, then the BFSR is not a universal form for two-column SM's. We shall show this by proving that the SM of Figure 3.3 is not binary CBFSR realizable even if one allows arbitrarily many codes per state.

	0	1
1	1	2
2	2	3
3	3	1

Figure 3.3

(By the techniques in Ref. 1, it is easy to show that the SM is not FSR realizable if under both inputs the FSR shifts right.)

The proof will be by contradiction. Let the FSR shift right under input 1 and left under input 0. Assume that one of the assignments for state 1 is $x_1x_2\dots x_k$. By the shift assumption and since state 1 maps to itself under input 0, it must also have the assignment $x_2x_3\dots x_k\bar{1}$. Since 1 maps to 2 under input 1 and since 1 has assignments $x_1x_2\dots x_k$ and $x_2x_3\dots x_k\bar{1}$, 2 must have assignment $\bar{x}_1x_2\dots x_k$. But, since 2 maps to itself under input 0, it must also have assignment $x_2\dots x_k\bar{1}$. Finally, since 2 maps to 3 under input 1, state 3 must have assignment $x_2\dots x_k$. But we assumed that 1 has assignment $x_1x_2\dots x_k$ and then showed that 2 has assignment $\bar{x}_1x_2\dots x_k$. Therefore

the assignment $Y x_2 \dots x_k$ for $\bar{3}$ must be either the assignment for 1 or

2. This finishes the proof.

It is interesting to note that if we only consider the CBFSR case, the above proof holds even if the successors of $\bar{3}$ are not specified, i.e., the flow table of Figure 3.4 is not CBFSR.

	0	1	
1	1	2	
2	2	3	
3	-	-	

Figure 3.4

If, however, we want to consider in this proof the unidirectional case, then both of the successors of $\bar{3}$ must be specified, as the flow tables of Figure 3.5 are unidirectional FSR realizable (with equivalent states).

	0	1		0	1
1	1	2	1	1	2
2	2	3	2	2	3
3	3	3	3	1	1

Figure 3.5

As a second restricted shift condition, let the shift direction be state determined such that all copies of a state shift the same direction. Under this assumption, the flow table of Figure 3.3 is again not BFSR realizable. Since we know that the flow table is not unidirectional FSR **realizable**, and because states 1, 2, and 3 have identical mapping properties, the only shift condition we must investigate is that in which states 1 and 2 shift right **and** state 3 shifts

left. Again, the proof will be by contradiction. Let one of the assignments for state β be $x_1 x_2 \dots x_k$. Then one of the assignments for state 1, from the shift assumption, must be $x_2 \dots x_k Z$. But 1 maps right to 1 and 2, and hence these must have assignments $Y x_2 \dots x_k$ and $\bar{Y} x_2 \dots x_k$, one of which must be the assignment for state β . This finishes the proof.

REFERENCES

1. Martin, R. L., "Studies in Feedback Shift-Register Synthesis of Sequential Machines," Report ESL-R-308, Electronic Systems Lab., M.I.T., Cambridge, Mass., May 1967.
2. Susskind, A. K., "Threshold Elements and the Design of Sequential Switching Networks," Technical Report No. RADC-TR-66-286, Section 4.3.
3. Hartmanis, J and R. E. Stearns, "Algebraic Structure Theory of Sequential Machines," Prentice Hall, Englewood Cliffs, N.J., 1966.

CHAPTER 4

REALIZATION OF SEQUENTIAL MACHINES IN A
CIRCUIT FORM CONTAINING A CLOCK

A clock is a sequential machine in which the state transitions are independent of the particular input symbols x_1, x_2, \dots, x_m . This definition appears to be meaningful only for the case of single-input machines, such as a counter. However, in the decomposition of a sequential machine M with more than one input symbol it is meaningful to say that "the machine M contains a clock". By this we mean that the machine can be realized by at least two submachines: the first submachine, M_C , is a clock, i.e., its operation is independent of the particular input symbol; the second submachine, M_B , is dependent on the state of the clock, the particular input symbol, and its own state. This leads to the decomposition shown in Figure 4.1.

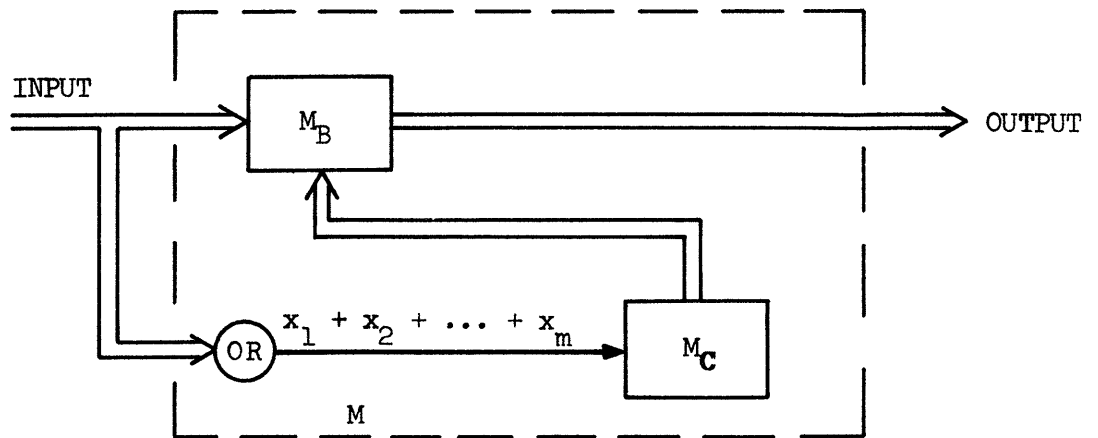


Figure 4.1 Decomposition of Machine M

This chapter describes how the submachine M_C , if it exists, can be found for any machine M without any trial and error whatsoever. Our motivation for investigating this particular circuit form arises from the fact that M_C is a single-input counter, which is simple to realize. For example, counters can be realized with feedback shift registers, or with one threshold element per state variable.

4.1 STATE DIAGRAM OF A CLOCK

Since a clock is a single-input machine, its state diagram must be a single cycle of one or more states into which lead directed branches from cycle-free subgraphs (i.e., trees). An example of this form is given in Figure 4.2(a). By appropriately combining states,

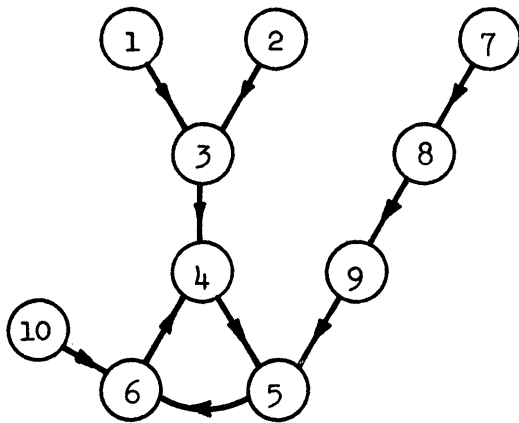


Figure 4.2(a) State Diagram

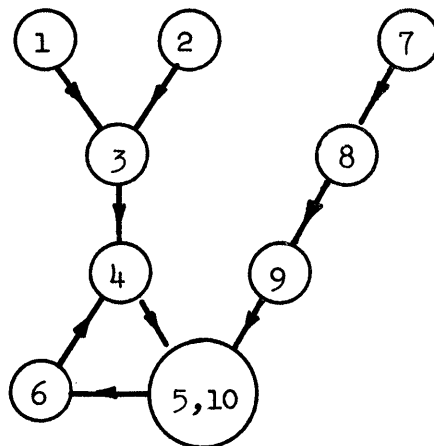


Figure 4.2(b) State 10 Absorbed

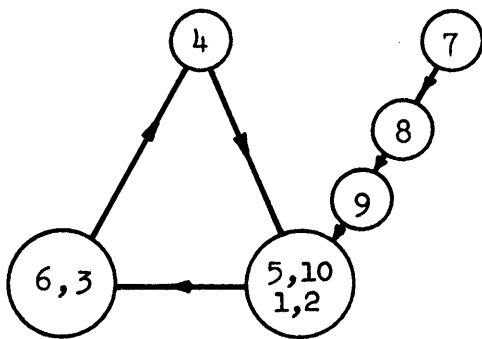


Figure 4.2(c) States 1, 2, and 3 Absorbed

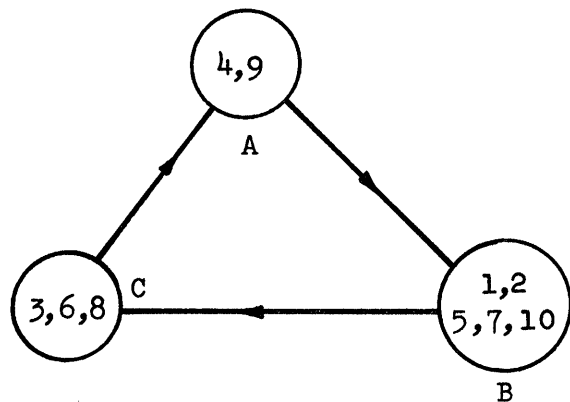


Figure 4.2(d) Final Form

the cycle-free subgraphs can be absorbed into the single cycle, as illustrated in Figures 4.2(b) through (d). In the final form, information has been lost. For example, we cannot tell whether state 4 goes to states 1,2,5,7, or 10. Figure 4.2(d) has as its nodes the three blocks $A = \{4,9\}$, $B = \{1,2,5,7,10\}$, and $C = \{3,6,8\}$. Every state appears in precisely one block and A maps to B, B to C, and C to A. Therefore, A, B, and C, and hence Figure 4.2(d), depict a "partition with the substitution property" (Ref. 1) on the machine of Figure 4.2(a).

Since the form of Figure 4.2(a) is general and the absorption process as illustrated can always be performed, we may conclude that the state diagram of every clock can be put into the form of a single cycle. (If the clock is a direct-sum machine, i.e., has two or more disconnected subgraphs, then it can be put in the form of one cycle for each disconnected part. The disjoint cycles can be combined, as will be shown later.)

Henceforth, it will be useful to think of the state diagram of a clock in its "absorbed" form, i.e., as a set of disjoint cycles without trees.

4.2 TEST FOR EXISTENCE OF A CLOCK

Lemma 4.1: (Hartmanis and Stearns) A machine M contains a clock iff there exists an input-independent partition on the states of M which has the substitution property, SP.

Proof: If M contains a clock, then that clock can be represented by a partition with SP, as illustrated previously. But the clock must be input independent, so that the partition must be input independent.

Conversely, if there exists an input-independent partition with SP, then that partition specifies a clock.

We now describe a novel method for efficiently finding the smallest possible input-independent partition with SP, if such a partition exists. We seek the smallest possible partition, P_C , so that the partition will have the largest possible number of blocks. This will hopefully make the task of the other component machine, M_B , as simple as possible, since M_B must contain as many states as there are elements in the largest block of P_C .

Partition P_C must be input-independent. Therefore, if states x and y are in the same successor row of M , they must be in the same block of P_C , and so states x and y must have the same assignment in the clock. Furthermore, if it is also true that states x and z are in the same successor row of M , then x and z must also have the same assignment in the clock. Consequently, x and y and z are in the same block of P_C . In terms of the notions and definitions of our previous report (Ref. 2, Section 3.1), this observation leads to the conclusion that

$$P_C \geq m^*$$

Applying the operator T repeatedly and recalling that P_C has the substitution property

$$\begin{aligned} P_C T &= P_C \geq m^* T \\ P_C T^2 &= P_C \geq m^* T^2 \\ &\vdots \\ P_C T^i &= P_C \geq m^* T^i \\ P_C T^{i+1} &= P_C > m^* T^{i+1} \end{aligned}$$

Recall that $m^{*T^{i+1}} \geq m^{*T^i}$ (Lemma 3.3 of the reference) and (Lemma 3.4) that there must exist some q such that either (a) $m^{*T^q} = (I)$, or (b) $m^{*T^q} = m^{*T^{q-1}}$. In case (a), $P_C = (I)$ and the "clock" has a single state, so that it is useless. In case (b), however, we can let $P_C = m^{*T^{q-1}}$. Then there will be as many states in the clock as there are blocks in $m^{*T^{q-1}}$. We have now demonstrated

Lemma 4.2: A given flow table has a nontrivial clock iff $m^{*T^q} = m^{*T^{q-1}} \neq (I)$. The longest clock period is found by assigning to each block of $m^{*T^{q-1}}$ one state in the clock.

We illustrate our procedure in Figures 4.3 and 4.4.

	x_1	x_2	
1	2	3	$m^* = (234, 15)$
2	3	4	$m^{*T} = (12345) = (I)$
3	5	5	
4	1	5	
5	3	3	

Figure 4.3 Flow Table Without Clock

	x_1	x_2	
a	b	e	$m^* = (be, cf, ah, i, d, g)$
b	c	f	$m^{*T} = (cfg, ah, be, i, d)$
c	h	h	$m^{*T^2} = (adh, be, cfg, i)$
d	i	i	$m^{*T^3} = (bei, cfg, adh)$
e	g	g	$m^{*T^4} = (cfg, adh, bei) = m^{*T^3}$
f	a	h	
g	d	d	
h	e	e	
i	c	c	

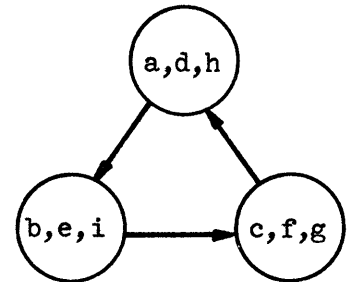


Figure 4.4 Flow Table With Clock of Period 3

4.3 AN ASIDE REGARDING FSR SYNTHESIS

If $m^*T^q = m^*T^{q-1}$, then we know that a clock exists. But then if we were to use m^*T^{q-1} as the basis for assigning state variable y_{q+1} in a feedback shift register (FSR) realization (i.e., $P_{q+1} \geq m^*T^{q-1}$), then we would obtain as the state-assignment partition for the preceding variable

$$P_q = P_{q+1}T^{-1} = P_{q+1}$$

This is true because $P_{q+1}T = P_{q+1}$, so that

$$P_q = P_{q+1}T^{-1} = P_{q+1}TT^{-1} = P_{q+1}$$

and a useless state assignment will be found because **all** state-assignment partitions will be identical. This leads to

Lemma 4.3: If M contains a clock, then there exists no state assignment in which state variables y_i and y_j are related by $y_i(t+1) = y_j(t)$.

Corollary: If M is FSR realizable, then there must exist a q such that $m^*T^q = (I)$.

4.4 USE OF EQUIVALENT STATES IN DECOMPOSITION

We return to the decomposition of Figure 4.1 and address ourselves to the following problem: if for a given machine M no clock can be found using Lemma 4.2, is there a way of introducing equivalent states so as to obtain the desired form? And if so, what is a simple procedure for determining equivalent states?

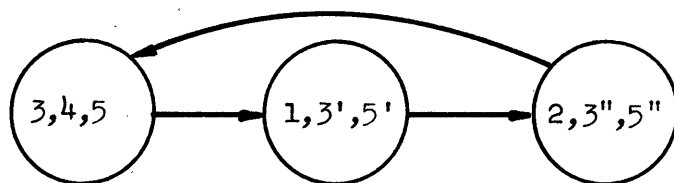
The answer to these questions is simple. Consider a row s in the flow table that has two or more distinct successors x, \dots, y .

Since s must be enumerated in at least one of the nodes, call it l , of the state diagram of the clock, states x, \dots, y must be together in node 2 , which is the successor to node 1 in the clock. Similarly, all the successors of all the states in node i must be in node $i+1$. If a clock exists, then there must be some node k (not containing all the states) which contains states all the successors of which are contained in node $j, j < k$. Then node k can be connected to node j and the resulting cycle represents the longest possible clock. Note that as we traced the successive states that could be reached from state s with any succession of inputs, nothing needed to be said about states appearing in the various nodes more than once. In fact, if state p appears in more than one node in the cycle, one need only differentiate between the various appearances of p by making these states equivalent to state p and calling them $p, p', p'',$ etc. The resulting clock describes the transitions of a machine with equivalent states which, when reduced, yields the original machine. Because the clock obtained has the longest possible cycle, only those equivalent states have been obtained which are necessary to achieve our objective.

We illustrate the procedure described above by reconsidering the machine specified in Figure 4.3, where previously it was found that without equivalent states no clock existed. Starting with state 1 , we obtain the following succession of mappings:

$$1 \rightarrow 2, 3 \rightarrow 3, 4, 5 \rightarrow 1, 3; 5 \rightarrow 2, 3, 5 \rightarrow (3, 4, 5)$$

Now distinguishing between the various appearances of 3 and 5 , we end up with the following clock:



This clock corresponds to the following machine, which can be reduced to that of Figure 4.3.

	x_1	x_2
1	2	3''
2	3	4
3	5'	5'
3'	5''	5''
3''	5	5
4	1	5'
5	3'	3'
5'	3''	3''
5''	3	3

The above construction leads to the longest possible clock cycle. If this requires more equivalent states than desired, one can reduce the cycle length by "folding" the cycle graph, providing its length is not a prime number. In folding in half, for example, nodes j and k are merged, as are $j+1$ and $k+1$, $j+2$ and $k+2$, etc. Of course, this is possible only if the cycle length is divisible by two. Folding into a third is possible if the length is divisible by three, etc. Since every integer can be written as a product of primes, it is easy to determine just how a given cycle can be folded.

We illustrate the reduction in equivalent states by folding for the machine of Figure 4.5. Starting with state 2, the following succession of

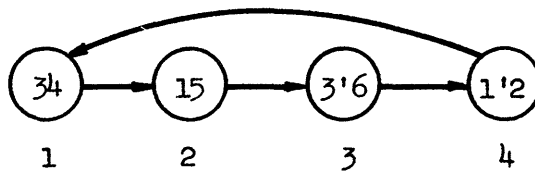
	x_1	x_2
1	3	3
2	3	4
3	1	1
4	1	5
5	6	3
6	1	2

Figure 4.5 Machine A

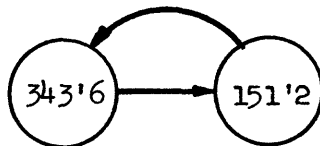
state sets is reachable from state 2:

$$2 \rightarrow 34 \rightarrow 15 \rightarrow 36 \rightarrow 12 \rightarrow (34)$$

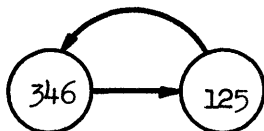
Hence the following clock is contained in A:



Merging nodes 1 and 3 folds the cycle in half:

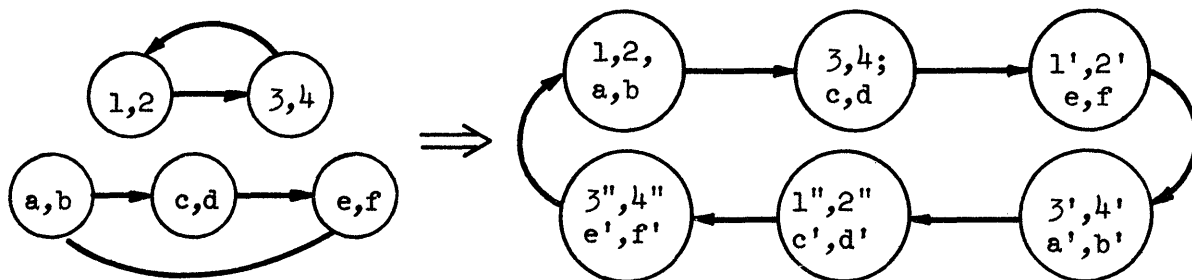


Now there is no point in carrying both 3 and 3', or 1 and 1'. So the following clock is obtained:



The cycle of length four (two) requires two (one) state variables in M_C and one (two) state variables in M_B . The former is probably preferable to the latter, because M_C is FSR realizable. (Note that any of the clocks are not only FSR realizable, but they can have particularly simple logic if the 1-out-of-N state assignment is used, where N is the number of nodes in the clock. Then, a rotating (or "end-around") shift register with N stages suffices, where the register advances every time any input occurs.)

The inverse of folding is required when the machine is a direct-sum machine. Then we find the clock for each sub-machine separately and merge these clocks in a single cycle of length equal to the least common multiple of the individual cycles. We illustrate this briefly below:

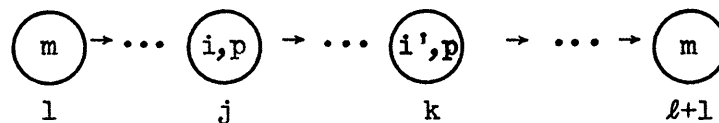


It is natural to inquire if the addition of equivalent states can make every machine decomposable into the form of Figure 4.1. The answer is negative, as was pointed out by R. L. Martin who showed

Lemma 4.4: If M is strongly connected and does not contain a clock, then there is no way of adding equivalent states such that the enlarged machine contains a clock.

To prove Martin's result, consider the states reached from state m . If there exists a shortest input sequence of length L such that m can return to m without having passed all other states, then there must exist other input sequences also of length L which let m reach the non-empty state set N . For if N were empty, then m could not reach all the other states, contrary to the assumption that M is strongly connected.

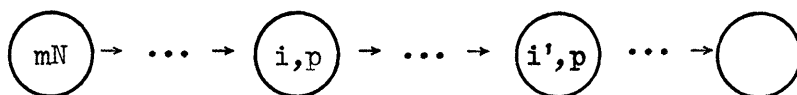
On the other hand, if m returns to m alone after passing through every one of the other states, then M contains a clock without equivalent states. For then the states intermediate between m and its re-appearance can be enumerated only once. If a state i were enumerated more than once, say in sets j and k , then we would have the following situation:



Here i' can reach m in less steps than are required by i , which contradicts that i and i' are equivalent.

Now return to the case where the first reappearance of m is accompanied by the state set N . Then the second reappearance of m could

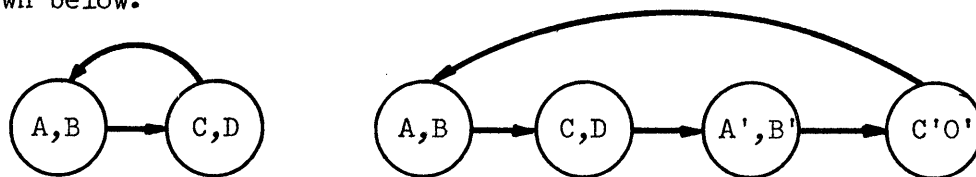
not be alone, but must be associated with at least N . If it is accompanied by precisely N , then a clock exists. If there were some state i enumerated twice between the two appearances of m , then the following clock would exist, where only the first node contains m . Because the



machine is strongly connected, both i and i' must have access to m . But then i' and i have shortest sequences leading to m which are different, and again i and i' cannot be equivalent. Thus every state can be enumerated only once.

This argument can be continued for the case where mN is followed by $mNO, mNOP, \dots$, where O, P, \dots are non-empty sets. Again either a clock exists with a single appearance of each state, or ultimately a reappearance of m is accompanied by all the other states. In the latter case, no clock can exist.

Of course, if a strongly-connected machine does have a clock, then that clock can be trivially extended by splitting each state, as shown below.



There is no value to this extension, unless it is used in bringing together disconnected sub-machines.

4.5 CONCLUSIONS

We have shown how a flow table can be tested for realizability in the form of Figure 4.1 by simple calculation of $m \cdot T^j$. For the case of strongly connected machines, this test quickly specifies the structure of the longest possible clock contained in the given machine and no equivalent states can be added to lengthen the clock. For the case of a machine that is not strongly connected, an equally simple technique was given for finding equivalent states that will permit decomposition involving a clock, if such a decomposition is possible. It is not necessary to determine whether or not the machine is strongly connected before applying the second technique. If the machine is strongly connected, then use of the second technique will also lead to the longest possible clock.

Finally, we have shown that if a flow table contains a clock, then that flow table cannot be feedback shift-register realizable. However, there exist cases where a flow table can be augmented by a set of equivalent states such that it becomes FSR realizable, and it can be augmented by some other (distinct) set of equivalent states such that it contains a clock.

REFERENCES

1. Hartmanis, J. and R. E. Stearns, "Algebraic Structure Theory of Sequential Machines," Prentice Hall, Englewood Cliffs, N.J., 1966.
2. Susskind, A. K., et. al., "Threshold Elements and the Design of Sequential Switching Networks," Technical Report No. RADC-TR-67-255.

SECTION III

FAULT DETECTION AND LOCATION IN COMBINATIONAL LOGIC

This section discusses various methods for finding good, i.e., short, tests that detect and/or locate faults in combinational logic. Chapter 1 (by R. J. Diephuis) treats fault detection in networks of AND, OR, NOT, and NAND gates. Chapter 2 (by J. M. Mazola and A. K. Susskind) deals with fault detection in single threshold gates. Whereas the work in Chapter 1 is restricted to single faults, that in Chapter 2 can handle arbitrarily many simultaneous faults. Finally, Chapter 3 (by A. K. Susskind) deals with a technique for finding tests that are applicable to both fault detection and fault location.

CHAPTER 1

AN ALGORITHM FOR FINDING APPROXIMATELY MINIMAL FAULT-DETECTION TEST SETS FOR COMBINATIONAL LOGIC

The algorithm to be described and discussed is based primarily on the ideas about fault detection by "path sensitizing" suggested by Armstrong [1], but uses a notation and calculus based on the "Calculus of D-Cubes" of Roth. [2,3] Parts of the contents of these two papers will be repeated here to make this report more-or-less self-contained. The approach here differs from Armstrong's in that it is not necessary to work with two-level AND-OR equations for a network (his "enf" and "complement enf"). The "T-cubes" resulting from the algorithm to be described represent information somewhat similar to the enf and complement enf, but it is believed that they contain more information and are more easily and efficiently computer-implemented than Armstrong's method. The method differs from that in a recent paper by Roth, et.al. [3] in the way the "D-cubes of sensitized paths" are used to find all those faults which a test vertex detects. Roth suggests first finding a test which detects an input fault, then applying an algorithm which determines what other faults this test detects - essentially by simulating the behavior of the network with this input applied for each possible fault. The method to be described uses D-cubes of other paths which have already been constructed to answer essentially the same question. Here the use of D-cubes also provides a means of finding those tests which detect many faults simultaneously; in Roth's algorithm, the vertices which are tried are chosen somewhat arbitrarily.

1.1 ASSUMPTIONS

1. Networks will be restricted to those containing AND, OR, NOT, NAND, and NOR gates. (The method can be modified to accept any gate which can be specified by a combinational function. It is felt, however, that the fault assumption on which the method is based is reasonable--at present--only for the types of gates listed here.)
2. Only single faults are considered.
3. All faults will be of the type which cause either an input or an output of a gate to appear to be "stuck at logical one" (s-a-1) or "stuck at logical zero" (s-a-0). A lead s-a-0 or s-a-1 need not cause other leads tied to it to be s-a-0 or s-a-1. In other words, the malfunction may be internal to a gate, and it need not affect other gates connected to its inputs; e.g., an open-circuited diode in a diode AND gate.
4. Only detectable faults will be considered. A fault is undetectable if its occurrence does not change the function realized by the network.

1.2 PATH SENSITIZING

The concept of path sensitizing, as described by Armstrong, will be discussed first. A portion of a network is shown in Figure 1.1a. Suppose it is desired to find a test which detects a s-a-1 fault at B, assuming that the path BCDEJ shown is the only path from B to an output. Then a test which detects a s-a-1 fault at B must be such that, when the circuit is normal, a 0 appears on B and 1's appear on all remaining inputs to NAND and AND gates in the path, and 0's

appear on all remaining inputs to NOR or OR gates in the path. This is necessary to insure that when B changes from 0 to 1 upon occurrence of the fault, the change propagates to output J where it can be observed as an improper output. The actual changes in values along the path BCDEJ upon occurrence of a fault at B are indicated in Figure 1.1a. The path BCDEJ is said to be "sensitized" when inputs are applied to the network which result in the assignments shown in Figure 1.1a.

It should be apparent from Figure 1.1a that a test for a s-a-1 fault on B will also detect a s-a-1 fault on J and s-a-0 faults on C, D, or E, since changes due to any of these faults will propagate to J. It should also be noted that to detect a s-a-0 fault on B, a test would be applied such that a 1 appears on B, while the other inputs to the gates on the path remain as before. (The path BCDEJ is still said to be "sensitized" - regardless of the value of lead B.) This test also detects faults along the path from B to J which are complementary to those detected by the s-a-1 test for B. Thus, the two tests together detect all s-a-0 and s-a-1 faults along the path BCDEJ.

The conditions stated above for a path to be sensitized may not hold on gates which are points of reconvergence for two or more fanout paths from some preceding gate, since two or more inputs to these gates may change simultaneously when a fault occurs. For example, in Fig. 1.1b, if an input is applied which results in the configuration shown, a s-a-0 fault on B causes changes on paths BGHEF and BCDEF. In this case, a multiple path is sensitized. Note that this test does not detect faults on lead C, D, G, or H. We will assume for most of

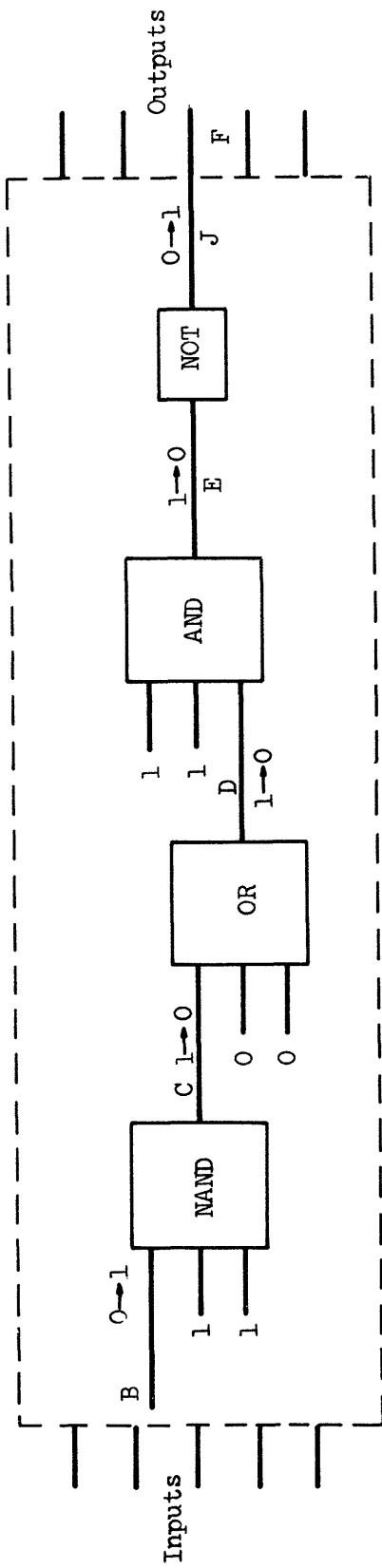


Fig. 1.1(a) A Portion of a Network With a Single Sensitized Path

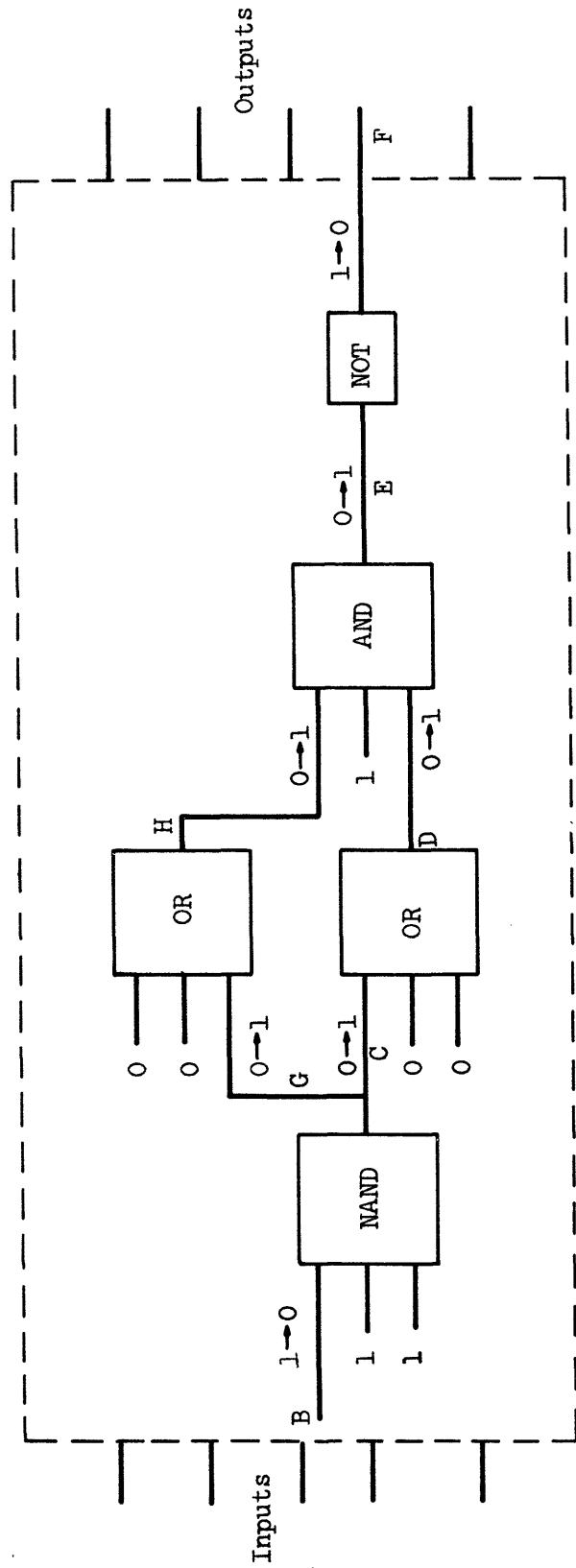


Fig. 1.1(b) A Portion of a Network With a Multiple Sensitized Path

this paper that all faults in the network to be analyzed can be detected without sensitizing multiple paths (see Section 1.7 and Appendix 1.1).

The above discussions can be made more precise by the following definitions:

lead (or connection) - A wire connecting the output of one gate to the input of another; or connecting an input variable to a gate input.

sensitized lead - The phrase "lead i is sensitized by vertex \bar{v} " is defined by the following experiment:

apply vertex \bar{v} to the network; disconnect lead i from the gate output or input variable to which it is normally connected; complement (the Boolean value of) lead i from what it normally is when \bar{v} is applied; if one or more of the network outputs is then different from its normal value when \bar{v} is applied, then lead i is sensitized by vertex \bar{v} .

Lead i is said to be sensitized for s-a-0 (s-a-1) by vertex \bar{v} if its normal value when \bar{v} is applied is 1 (0) (since \bar{v} will detect a s-a-0 (s-a-1) fault on lead i).

path - An ordered set of leads (l_1, l_2, \dots, l_n) such that for all i , $1 \leq i < n$, l_i is a gate input lead to the gate for which l_{i+1} is a gate output lead.

sensitized path - A path leading from lead i to an output of the network is said to be sensitized by vertex \bar{v} if, when the above experiment is performed for lead i , complementing i

causes the value of all leads on the path to be complemented. If, for the particular vertex \bar{v} , there is only one sensitized path from lead i to some output, we will say that a single path is sensitized from i to this output. If two or more paths from lead i to the same output are sensitized by \bar{v} , we will say that a multiple path is sensitized by \bar{v} from i to this output. The term "sensitized path" from a lead to an output with no qualifying adjectives will always refer to a single sensitized path.

The above remark that, in Figure 1.1a, the test for lead B also detects other faults on the (single) sensitized path can be stated as:

Theorem 1.1: If lead i is sensitized by vertex \bar{v} , then all leads on a single sensitized path from i to an output are also sensitized by \bar{v} .

One way to find small test sets is to find tests, each of which detects many different faults. In light of the above observations, one expects that "good" tests are those which sensitize long paths, i.e. paths containing many connections. The longest sensitizable paths in the network are from the inputs to the outputs; in fact, the following obvious theorem can be stated:

Theorem 1.2: A set of vertices V detects all single s-a-0 and s-a-1 faults in the network if it has the following properties:

- a) for all inputs i to the network, there exists $\bar{v} \in V$ such that \bar{v} sensitizes input i for s-a-0, and there exists $\bar{v}' \in V$ such that \bar{v}' sensitizes input i for s-a-1.

- b) for all leads l in the network, there exists $\bar{v} \in V$ such that lead l is 0 when \bar{v} is applied and lead l is on a single path sensitized by \bar{v} ; and there exists $\bar{v}' \in V$ such that lead l is 1 when \bar{v}' is applied and lead l is on a single path sensitized by \bar{v}' .

In other words, the set of paths sensitized covers all connections in the network for both s-a-0 and s-a-1 faults.

If the network contains no fan-out, then only condition a) is needed, because in this case there is only one path from each input to the outputs, and the set of paths sensitized by tests for the net inputs must cover all connections. Note that the above theorem says nothing about the existence of a set of tests satisfying a) and b). This will be discussed further in Section 1.7.

It has been observed, by trying many examples, that tests usually exist which simultaneously sensitize several single paths through a network. For example, in Figure 1.2, if inputs can be chosen so that the configuration shown results, paths BCDEJ and RSTTEJ are both

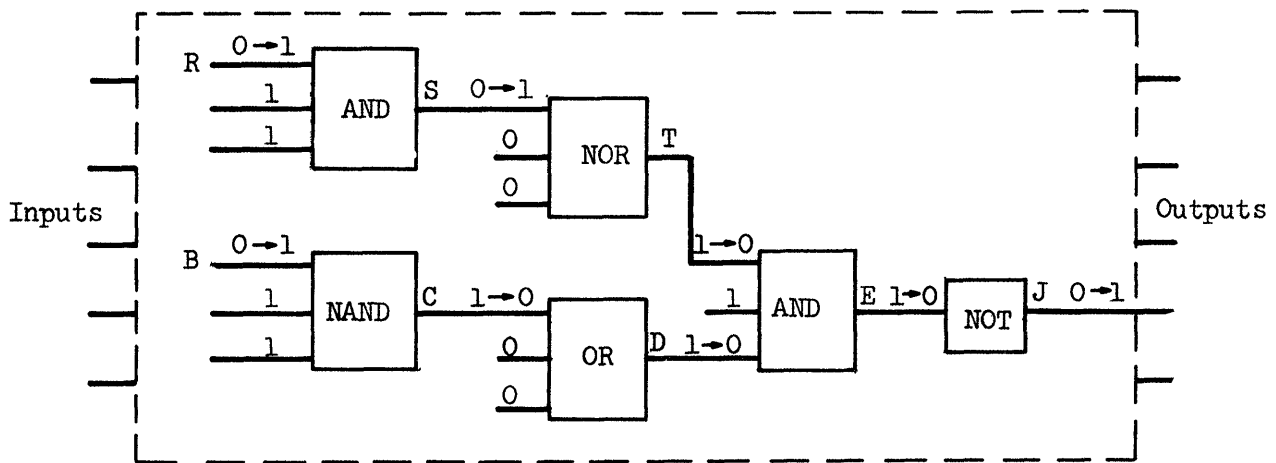


Fig. 1.2 Network With Two Sensitized Single Paths

sensitized, and a single test will detect a single fault in any of these connections (since both of these paths are single paths).

The method to be described finds tests each of which detect a single fault on many leads by first finding the conditions necessary to sensitize each possible single path from an input to an output, and then combining these conditions into tests which sensitize many single paths simultaneously. (This procedure must be modified to include multiple paths for networks that have no test set satisfying a) and b) above. This will be discussed further in Section 1.7.)

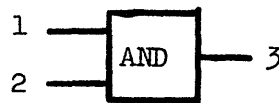
1.3 NOTATION

The notation to be described here is a modification of the "calculus of D-cubes" of Roth. The notation serves three different purposes:

- a) specifying the structure and behavior of the network;
- b) representing sensitized paths;
- c) representing a test and the connections which it tests.

1.3.1 Specification of the Network Behavior - The Singular Cover

The network will be specified by its "singular cover", to be described below. Consider the simple AND circuit shown below:



To specify its behavior, we must know for what inputs the output is 0, and for what inputs it is 1. This is done most concisely by specifying "cubes" for which it is 0 or 1, rather than an entire truth table. The singular cover for the AND block above is shown below:

1	2	3
1	1	1
0	x	0
x	0	0

The interpretation of this is as follows: output line 3 is 1 only when inputs lines 1 and 2 are 1; line 3 is 0 when input 1 is 0 (regardless of the value of input 2 - the x stands for either 0 or 1) or when input 2 is 0 (regardless of input 1). Thus

1	2	3
1	1	1

stands for a cube containing the single vertex 111. However,

1	2	3
x	0	0

stands for a cube containing the two vertices 000 and 100.

A cube of a gate's singular cover is said to be "prime" if it is contained in no other cube of the cover and there is not an "x" on the output lead (i.e., it corresponds to a prime implicant of the function or its complement). We will use here only cubes of the singular cover of a gate that are prime. Hence the singular cover may be defined as the set of all prime cubes for a gate.

The singular cover of a network is just the set of singular covers of each of its gates. The network in Figure 1.3 has the singular cover shown below it.

Note that a "blank" and an x have the same meaning in the above table. The x's simply emphasize those lines of the gate from which a particular section of the table is derived. E.g., the cube b can be written:

1	2	3	4	5	6	7	8	9	10	11
x	1	x	1	1	x	x	x	x	x	x

The interpretation of each of these cubes is exactly as for the one-gate case. E.g., the cubes of lines d, e, and f indicate that to have a 1 on line 9, there must be 1's on lines 3, 4, and 5. For a zero on line 9, there must be a 0 on either line 3 or lines 4 and 5. Note that all connections (i.e., a wire between two gates) or leads are given distinct numbers - this is because we want to allow each connection to be independently s-a-0 or s-a-1. In Figure 1.3, the singular covers of each gate have been enclosed in dotted lines. The entries in the table outside these lines are caused by one of two conditions: (1) two or more connections are the output of the same gate and normally must have the same signal, or (2) two or more connections connect to the same input variable or complemented input variable. In other words, the values shown for these lines are "forced" by the values in the cube of the singular cover of the gate.

In summary, the algorithm for finding the singular cover of a network from the singular cover of its gates is as follows:

1. Number all connections. (It is convenient to do this so the numbers of all input connections to any gate are smaller than the numbers of any of its output connections.)
2. Enter the singular cover of each gate on separate rows in the table.
3. For each row, enter the values for any connections which are "forced" by those already assigned due to connection to a common input variable or to a common output.

1.3.2 Representation of a Sensitized Path - D-Cubes

A sensitized path will be represented by its D-cube. The D-cube specifies both the path which is sensitized and the assignments to other connections which cause the path to be sensitized. Suppose, for example, that the path through connections 2-4-9-11 of the network in Figure 1.3 is to be sensitized. For this to occur, connection 10 must be 0, connection 3 must be 1, and connection 1 must be 0. This is specified by the D-cube:

1	2	3	4	5	6	7	8	9	10	11
0	D	1	D	B				D	0	D

The D's indicate the connections in the path which is sensitized, and the B's indicate connections at the "edges" of the path where changes occur, but do not necessarily propagate to an output (i.e., for the above cube, line 5 changes from 0 to 1 if line 4 does, but line 5 is not on the sensitized path). The D-cube can be interpreted as follows: If a connection marked with a D is 0 (1), then all connections on the path from it to the output marked with a D or B are 0 (1). The reason for having B's in addition to D's is to specify those connections not on the sensitized path which must change in value when the values of connections on the path are changed.

This D-cube is a "cube" in the sense that it contains certain vertices of the 11-dimensional space of all binary 11-dimensional vectors. Note that the vertices included in a D-cube do not actually form a single cube in n-dimensional space. The D-cube shown above contains the 2^4 vertices in the cubes:

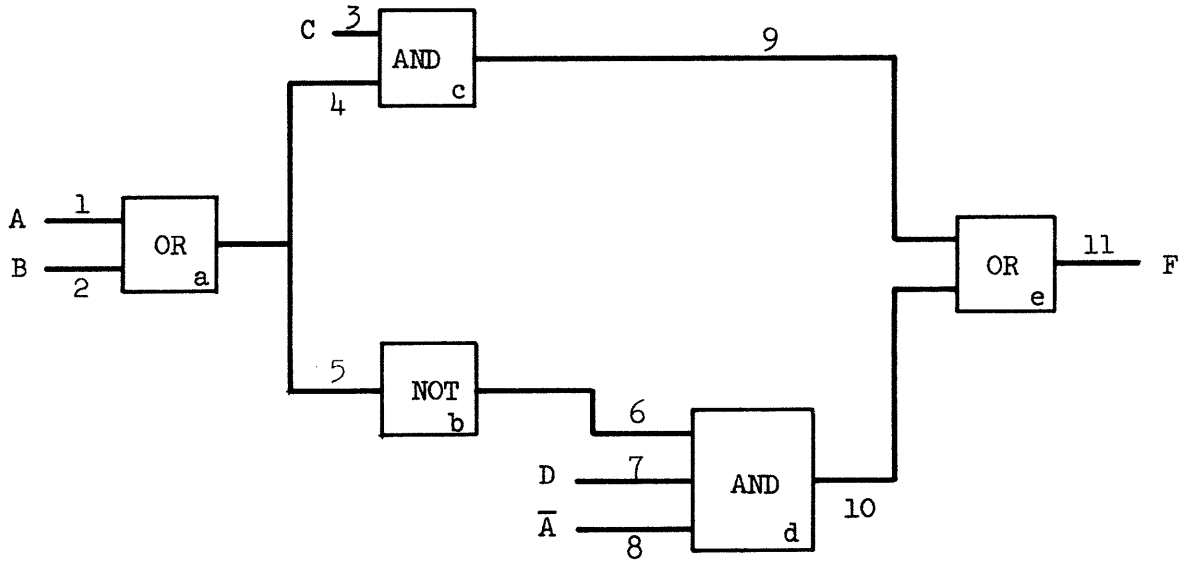


Fig. 1.3(a) Network

	1	2	3	4	5	6	7	8	9	10	11
a	0	0		0	0			1			
b	x	1		1	1			x			
c	1	x		1	1			0			
d			1	1	1				1		
e			x	0	0				0		
f			0	x	x				0		
g				0	0	1					
h				1	1	0					
i	0					1	1	1		1	
j	x					0	x	x		0	
k	x					x	0	x		0	
l	1					x	x	0		0	
m									0	0	0
n									x	1	1
o									1	x	1

(b) Its Singular Cover

Fig. 1.3 Running Example

1	2	3	4	5	6	7	8	9	10	11
0	1	1	1	1	x	x	x	1	0	1
0	0	1	0	0	x	x	x	0	0	0

The path specified by the D's in this D-cube is said to be sensitized because a change from 0 to 1 (1 to 0) in a connection marked with a D causes a 0 to 1 (1 to 0) change on all connections marked with a D on the path from it to an output.

Note that the above D cube is not "complete" in the sense that the conditions for the path to be sensitized are not specified in terms of controllable input variables only. E.g., connection 10 is required to have a 0, but the specifications for leads 6, 7 and 8 for this to occur is not given. One way to "complete" this D cube is to require the input on lead 7 to be 0 (note that lead 8 is already specified as 1, since leads 1 and 8 are connected to the same input variable). This gives the "complete" D-cube:

1	2	3	4	5	6	7	8	9	10	11
0	D	1	D	B	x	0	1	D	0	D

Thus, assigning input variables $A = 0$, $C = 1$, $D = 0$ sensitizes the path 2-4-9-11. Setting input variable $B = 0$ gives a test for connections 2, 4, 9, and 11 s-a-1, while setting $B = 1$ gives a test for these connections s-a-0.

Now suppose that the path 1-5-6-10-11 is to be sensitized. It is easily seen that this is done by the complete D-cube:

	1	2	3	4	5	6	7	8	9	10	11
$D_a =$	D	0	0	B	D	\bar{D}	1	1	0	\bar{D}	\bar{D}

where a \bar{D} indicates that a connection so marked is 1 (0) when a connection before it in the path marked with D is 0 (1). Note that

the meaning of D and \bar{D} is only relative to other D's or \bar{D} 's in a D-cube.

E.g., the D-cube

$$D_b = \begin{array}{cccccccccccc} \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline \bar{D} & 0 & 0 & \bar{B} & \bar{D} & D & 1 & 1 & 0 & D & D \end{array}$$

has exactly the same meaning (and includes the same vertices - in this case only two) as the D-cube D_a . These D-cubes are complements of each other (written $D_a = \bar{D}_b$) and are said to be equivalent since they contain the same vertices.

In order to sensitize the path 1-5-6-10-11, we were forced to assign connection 8 to 1, thus requiring input variable A to be 0. But setting A = 0 also assigns a 0 to connection 1, which is on the path to be sensitized. This means that the path 1-5-6-10-11 can only be sensitized to detect s-a-1 faults on connection 1 (since to detect s-a-0 faults on lead 1, lead 1 must be assigned with a 1; but this causes lead 8 to become 0, and the path is no longer sensitized). The fact that the path can be sensitized only when D = 0 for lead 1 is indicated by putting a subscript of 0 on the D for the lead for which the restriction occurs, so the D-cube of this path should be:

$$D_a = \begin{array}{cccccccccccc} \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline D_0 & 0 & 0 & B & D & \bar{D} & 1 & 1 & 0 & \bar{D} & \bar{D} \end{array}$$

This D-cube is interpreted as follows: If lead 1 is 0 (with other leads assigned constants as specified in the cube), then all leads with a D or B are 0, and all leads with \bar{D} or \bar{B} are 1. If lead 1 could be made 1 without changing the constants on other leads (e.g., by a fault), then all leads with a D or B (\bar{D} or \bar{B}) would be 1 (0).

The D-cube itself contains only one vertex:

1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	1	1	1	1	0	1

since this is the only vertex consistent with it and the properly operating network. \bar{D}_a is formed by complementing all the D's and B's (but not changing constants or subscripts). We will write \bar{D}_0 , but this should be interpreted as meaning $(\bar{D})_0$ - i.e., the lead must be assigned a value of 0; the \bar{D} only specifies how the value of this lead is related to the value of other leads in the cube. Thus, \bar{D}_a is:

	1	2	3	4	5	6	7	8	9	10	11
$\bar{D}_a = \bar{D}_0$	0	0	B	D	D	1	1	0	D	D	

A somewhat different situation occurs when a connection which is not an input lead marked with a B or D must have a constant value for a path to be sensitized. Suppose that to sensitize the path 2-4-9-11, we wished to assign lead 7 and 8 to be 1 (i.e., A = 0, D = 1). Then for lead 10 to be 0, lead 6 must be 0, in turn requiring lead 5 to be 1 (because of the inverter). Using the subscripting convention described above, this gives the D-cube:

	1	2	3	4	5	6	7	8	9	10	11
$D_c =$	0	D	1	D	B ₁	0	1	1	D	0	D

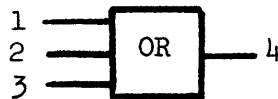
This implies that the cube is valid only for D = 1. But suppose that the input assignment specified by D_c is applied to the network (A = 0, B = 1, C = 1, D = 1). One can see from examining the network that if lead 2 were to become 0 due to a fault, the output would remain 1. (This is because the path 5-6-10 is actually also sensitized, and the inversion in this path causes one of the inputs to the final OR gate to be 1 regardless of the value of lead 2). Thus, while the above

D-cube may make sense "abstractly", it is not useful for finding sensitized paths which detect s-a-0 and s-a-1 faults because lead 5 does not remain 1 when a fault on lead 2 occurs. (This is not completely true, since the input vertex specified by this cube could still detect s-a-0 faults on leads 4, 9, and 11; however, we are only considering tests which test entire paths from an input to an output.)

For this reason, we will not use cubes which contain subscripted B's or D's on other than input leads in finding test sets. (This is not completely justified for networks with reconverging paths with no inversion in either path. This problem is considered further in Section 1.7.) In addition, if it is desirable to detect faults due to an input variable (as opposed to an input lead) s-a-0 or s-a-1, then no cubes with subscripted B's or D's should be used.

1.3.3 Primitive D-Cubes

The algorithm to be described starts with the set of primitive D-cubes of each logical element, and, from these, builds up D-cubes of paths through the network. The set of primitive D-cubes of a logical block is simply those D-cubes specifying all the single sensitized paths through that element. E.g., the 3-input OR gate below



has the following primitive D-cubes:

1	2	3	4
D	0	0	D
0	D	0	D
0	0	D	D

Similarly, a 3-input AND gate



has primitive D-cubes

5	6	7	8
D	1	1	D
1	D	1	D
1	1	D	D

(The primitive D-cubes of a logical block can be obtained from its singular cover using a simple construction described in Roth^[2].)

The primitive D-cubes of a network is just the set of primitive D-cubes for each block in the network. A "B" is used rather than a D for the output leads of each gate which is not a network output. (As will be seen later, one of these B's is replaced by a D when the lead becomes part of a path leading toward an output of the network. Those output leads not on a path remain as B's.) As in the construction of the singular cover of a network, we also specify values for those connections whose values are "forced" to be constants by assignments in a primitive D-cube. A "B" is used when the value of a lead must be the same as that of a lead to which a "D" is assigned by the primitive D-cube. The primitive D-cubes of the network in Figure 1.3 are given below:

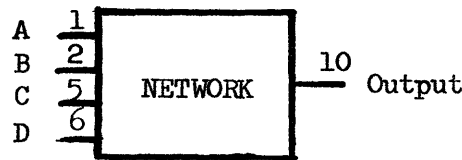
	1	2	3	4	5	6	7	8	9	10	11
a	0	D		B	B			1			
b	D	0		B	B			\bar{B}			
c			1	D	B				B		
d			D	1	1					B	
e				B	D	\bar{B}					
f	0					D	1	1		B	
g	0					1	D	1		B	
h	\bar{B}					1	1	D		B	
i									0	D	D
j									D	0	D

1.3.4 Representation of a Test - T-Cubes

A test for a group of faults in a network will be represented by a T-cube. The T-cube indicates both the input assignment for the test and all the single faults that it detects. (Again, this does not in general mean that the test will detect two or more faults occurring simultaneously, but rather that an improper output will occur if any one of the specified faults occurs.) A typical T-cube is:

1	2	3	4	5	6	7	8	9	10
T	0	1	\bar{T}	x	\bar{T}	\bar{T}	T	T	\bar{T}

A "T" indicates that this test will detect a s-a-0 fault on this lead and that the lead is assigned to be 1 for this test. A " \bar{T} " means that this test detects s-a-1 fault, and the lead must be assigned to be 0. A "0" or "1" means the lead must be assigned to be 0 or 1, but a fault on this lead will not necessarily be detected. Symbol "x" means that the lead can be assigned 0 or 1, but a fault is not necessarily detected. If the above T-cube is a test for the network



it means that input vertex $A = 1, B = 0, C = 0$ or $1, D = 0$ detects s-a-0 faults on leads 1, 8, and 9 and s-a-1 faults on leads 4, 6, 7, and 10.

1.3.5 The Operation of Intersection

In the algorithm to be described, it is often necessary to find the vertices which two D-cubes have in common, i.e., those vertices which are consistent with the assignments specified by both D-cubes. If the D-cubes are considered as representing sets of vertices, this is an intersection operation. As an example, consider the following two D-cubes

	0	1	2	3	4	5	6	7	8	9
$D_a = 0$	D	1	D	x	\bar{D}	0	x	1	x	
$D_b = 0$	x	1	D	0	\bar{D}	0	D	D	\bar{D}	

One way to find $D_a \cap D_b$ would be to list the vertices contained in each cube, take the intersection of these sets, and then construct a D-cube for the resulting set. It is, however, much easier to do the intersection component by component. (Each symbol in a D-cube is called a component.) For example, examining the components for lead 0 tells us that any vertex in the intersection must have a 0 on lead 0. Similarly, any component which is a constant in both cubes must also be the same constant in $D_a \cap D_b$. If any components have different constants in each cube, then $D_a \cap D_b$ is empty, denoted by ϕ . Examining those leads with D's in both cubes (ignore for the moment leads 8 and 9), the D's in D_a indicate that if lead 1 is 0 (1), then lead 3 is 0 (1) and lead 5 is 1 (0), and those in D_b that if lead 3 is 0 (1) then lead 5 is

1 (0) and lead 7 is 0 (1). Clearly this implies that in the intersection, if lead 1 is 0 (1), then leads 3 and 7 are 0 (1) and lead 5 is 1 (0). Thus components 0 to 7 of $D_a \cap D_b$ must be:

0	1	2	3	4	5	6	7
0	D	1	D	0	\bar{D}	0	D

Now consider lead 8. This component of the intersection must be D_1 , since D_a requires this lead to be 1, but if this lead could become 0 without changing other constant assignments, the other D's in D_b could still change (all D's between lead 8 and the output would change). Thus

$D_a \cap D_b$ is:

0	1	2	3	4	5	6	7	8	9
0	D	1	D	0	\bar{D}	0	D	D_1	\bar{D}

Now suppose the intersection of the following cubes is desired

	0	1	2	3	4	5	6	7	8	9
$D_a =$	0	D	1	D	x	\bar{D}	0	x	1	x
$\bar{D}_b =$	0	x	1	\bar{D}	0	D	0	\bar{D}	\bar{D}	D

Certainly the intersection is the same as found above, since we have seen that $D_b = \bar{D}_b$. To use the component-by-component intersection rules described above, we must first complement \bar{D}_b , and then proceed as before. In general, in forming the intersection we will always try to intersect component by component with the cubes given - if a D appears as the i^{th} component for one, and a \bar{D} for the other, then we complement one of the cubes and try again. If a D and \bar{D} still appear on a common component, then the intersection of the cubes is empty.

One final complication occurs when both cubes contain D's or \bar{D} 's, but there is no component which is D or \bar{D} in both cubes. E.g.

$$\begin{array}{cccccc}
 & & & 1 & 2 & 3 & 4 & 5 \\
 & & & \hline
 D_c = & 0 & D & x & \bar{D} & x \\
 D_d = & 0 & x & D & x & D
 \end{array}$$

Following the above ideas gives:

$$\begin{array}{cccccc}
 & & & 1 & 2 & 3 & 4 & 5 \\
 & & & \hline
 D_c \cap D_d = & 0 & D & D & \bar{D} & D
 \end{array}$$

but this does not include all vertices which are in the set $D_c \cap D_d$. The problem is that the D's from the two cubes are really independent - e.g., it is consistent with both of the cubes for lead 2 = 0, lead 3 = 1, lead 4 = 1, lead 5 = 1. This, however, is not allowed by the cube for $D_c \cap D_d$ obtained as above. This problem can be resolved in several ways (e.g., using different symbols for the D's of the two cubes, or by using two D cubes to describe the intersection), but, for the algorithm to be described, this situation will never occur. Thus, we will just say that the D-intersection is undefined when the above situation occurs. This is reflected in rule 3 given below. (Note that $D_i \cap D_j$ is undefined iff $D_i \cap D_j \neq \phi$ and $D_i \cap \bar{D}_j \neq \phi$.)

The rules for intersection of B's on a common component are the same as those for D's. The intersection of a D (\bar{D}) and a B (\bar{B}) is defined to be a D (\bar{D}), since, as will be seen in section 1.4.1, this situation occurs when a B on one of the output leads of a gate is being "linked" with a D on an input component of a D-cube of another gate - thus forming a sensitized path through one more gate in the network. [It should be noted that the B's are really serving two different, but related, purposes: (1) They indicate input leads which are connected to the same input variable as an input lead on the sensitized path.

Thus assigning the D's (\bar{D}) in such a D-cube to be 0 or 1 (1 or 0) also requires the input leads marked with B (\bar{B}) to be assigned 0 or 1 (1 or 0). (2) They, in a similar manner, indicate those outputs of a gate on a sensitized path which are not part of the sensitized path, but which clearly must change when the leads on the sensitized path change.]

In order to formally define D-intersection, we will change slightly our way of writing D-cubes. Each component of a D cube will have the following form:

$$(E)_k$$

where

$$E = D, \bar{D}, B, \bar{B}, \text{ or } x$$

$$k = 0, 1, \text{ or } x$$

The relation between this notation and our previous notation should be obvious. E.g., the D-cube

$$\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & D & 1 & B_0 & \bar{D}_1 & \bar{B} & x \end{array}$$

would be written in the new notation as:

$$\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline (x)_0 & (D)_x & (x)_1 & (B)_0 & (\bar{D})_1 & (\bar{B})_x & (x)_x \end{array}$$

The subscripts specify all constant leads, while the term inside () specifies the sensitized path.

Also, we need to define

$$E' = \begin{cases} D (\bar{D}) & \text{if } E = D (\bar{D}) \\ D (\bar{D}) & \text{if } E = B (\bar{B}) \\ x & \text{if } E = x \end{cases}$$

and

$$\bar{E} = \begin{cases} \bar{D} (D) & \text{if } E = D (\bar{D}) \\ \bar{B} (B) & \text{if } E = B (\bar{B}) \\ x & \text{if } E = x \end{cases}$$

With these definitions, D-intersection can be formally defined as follows:

Let $A = a_1 a_2 \dots a_n$ where $a_i = (E_i)_{k_i}$

$B = b_1 b_2 \dots b_n$ where $b_i = (F_i)_{l_i}$

Then $A \cap B$ is found as follows:

1. Let $c_i = (E_i * F_i)_{k_i l_i}$

$d_i = (E_i * \bar{F}_i)_{k_i l_i}$

where

$$E_i * F_i = \begin{cases} E_i & \text{if } F_i = x \\ F_i & \text{if } E_i = x \\ E_i & \text{if } E_i = F_i \\ E'_i & \text{if } E'_i = F'_i; E_i \neq F_i \\ \phi & \text{otherwise} \end{cases}$$

$$k_i l_i = \begin{cases} k_i & \text{if } l_i = x, \text{ or } k_i = l_i \\ l_i & \text{if } k_i = x \\ \phi & \text{otherwise} \end{cases}$$

$c_i = \phi$ if $E_i * F_i = \phi$ or $k_i l_i = \phi$

$d_i = \phi$ if $E_i * \bar{F}_i = \phi$ or $k_i l_i = \phi$

2. If there exist i and j such that

$c_i = d_j = \phi$, then $A \cap B = \phi$

3. If there exists no i such that

$c_i = \phi$ or $d_i = \phi$, then $A \cap B$ is undefined.

4. If neither 2 or 3 are satisfied, then

$$A \cap B = c_1 \dots c_n \quad \text{if there exists no } i \text{ such that } c_i = \phi$$

$$A \cap B = d_1 \dots d_n \quad \text{if there exists no } i \text{ such that } d_i = \phi$$

It should be noted that the operation of D-intersection is both commutative and associative when it is defined, assuming that complement D-cubes are considered to be equal.

1.4 CONSTRUCTION OF D-CUBES FOR SENSITIZED PATHS OF A NETWORK

The construction of D cubes for sensitized paths of a network starting with the primitive D-cubes and singular cover of the network is done in two steps:

1. Formation of an incomplete D-cube for the path from the primitive D-cubes.
2. Completing the D-cube using the singular covers of the network (called the "consistency operation" in Roth).

These steps will be described in detail below, using the network of Figure 1.3 as an example.

1.4.1 Formation of Incomplete D-cubes

The formation of an incomplete D-cube of a path consists simply of "linking together" the primitive D-cubes of each logical block along the path. This is best explained by again considering the network of Figure 1.3. Again, we will consider the paths 2-4-9-11 and 1-5-6-10-11. Path 2-4-9-11 passes through gates a, c, and e; so we must consider the primitive D-cubes of these blocks. Examining the primitive D cubes for the network (see Section 1.3.3), we see that to sensitize the path 2-4 we must use primitive D-cube 'a of gate a; to sensitize the path 4-9, we must use D-cube c of gate c; and to sensitize path 9-11, use D-cube j

of gate e. Because the D-cube of the entire path must contain only those vertices which are consistent with the appropriate primitive D-cubes of all gates on the path, these three cubes are "linked" together by simply combining them into one D-cube using the intersection operation defined above. (The intersection will always be defined, since we always choose D-cubes which have a D and a B on the same lead.) Applying the above to first form $D_a \cap D_c$ then $D_a \cap D_c \cap D_j$, we get the incomplete D-cube:

$$D_{acj} = \begin{array}{cccccccccccc} \hline & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline & 0 & D & 1 & D & B & & & 1 & D & 0 & D \end{array}$$

(The subscripts indicate the cubes which were intersected to form D_{acj} .)

Using the same ideas for path 2-5-6-10-11, we choose D-cubes b, e, f, and i of gates a, b, d and e and intersect them to form:

$$D_{befi} = \begin{array}{cccccccccccc} \hline & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline & D_0 & 0 & & B & D & \bar{D} & 1 & \bar{B}_1 & 0 & \bar{D} & \bar{D} \end{array}$$

In summary, to form the D-cube of a path:

1. For each logic block along the path, choose the primitive D-cube which has a D for the input which is part of the path.
2. Link all of these D-cubes together using the operation of intersection.

1.4.2 Completing the D-Cubes

The D-cubes found by the above method are not complete in the sense that the conditions for the path to be sensitized are not specified in terms of input variables. In order to specify these conditions in terms of input variables, the incomplete D-cube must be intersected with the singular covers of the gates not on the sensitized paths.

(Again, we want to find those vertices which are consistent with both the incomplete D-cube and one of the prime cubes of each gate not on the path.) This will be illustrated with the example of the previous section.

Consider the incomplete D-cube:

$$D_{acj} = \begin{array}{cccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline & 0 & D & 1 & D & B & x & x & 1 & D & 0 & D \end{array}$$

It is usually convenient to work from right to left in the D-cube, determining what conditions are necessary on other leads to produce the conditions necessary on leads already specified. Thus, we start with lead 10, which must have value 0 for the path to be sensitized. Since this is the output of gate d, the conditions under which this lead is 0 are specified by the singular cover of gate d. From the singular cover given in Figure 1.3, we see that any of the following cubes result in a 0 on lead 10:

$$\begin{array}{cccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline j & & & & & & 0 & x & x & & 0 & \\ k & & & & & & x & 0 & x & & 0 & \\ \ell & & & & & & x & x & 0 & & 0 & \end{array}$$

Thus, one of these three cubes must be intersected with the incomplete D-cube. Intersecting D_{acj} with cubes j and k gives two new D-cubes:

$$\begin{array}{cccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline D_{acj}^j & = & 0 & D & 1 & D & B & 0 & x & 1 & D & 0 & D \\ D_{acj}^k & = & 0 & D & 1 & D & B & x & 0 & 1 & D & 0 & D \end{array}$$

(Superscripts indicate the singular covers used in forming a D-cube.)

D_{acj} cannot be combined with D_{acj}^ℓ since the components representing lead 8 give $1 \cap 0 = \phi$.

Consider now D_{acj}^j . The 0 on lead 10 has now been specified in terms of the input to the gate previous to it, but the components to the left of 10 must still be checked. Lead 8 is an input lead, so the 1 on it is specified in terms of inputs. However, lead 6 is not an input lead; it is the output of gate b. Thus the singular cover of gate b must be inspected to find the conditions on its input for lead b to be 0. Only cube h in the singular cover gives a zero on lead 6, so it must be combined with D_{acj}^j ; giving:

$$D_{acj}^{jh} = \begin{array}{cccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline & 0 & D & 1 & D & B_1 & 0 & x & 1 & D & 0 & D \end{array}$$

Continuing as above, we see that leads 3 and 1 are input leads, therefore D_{acj}^{jh} is a complete D-cube. However, as was seen in Section 1.3.2, because this cube has a component which is B_1 and is not an input lead, it is not a cube for the entire sensitized path 2-4-9-11, i.e., the output does not change if lead 2 changes from 1 to 0.

Similarly, D_{acj}^k can be seen to be complete. This can be seen to be the only complete D-cube corresponding to the sensitized path 2-4-9-11:

$$D_{acj}^k = \begin{array}{cccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline & 0 & D & 1 & D & B & x & 0 & 1 & D & 0 & D \end{array}$$

In the same way, the D-cube for path 2-5-6-10-11 can be completed by combining it with cube f of the singular cover, giving:

$$D_{befi}^f = \begin{array}{cccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline & D_0 & 0 & 0 & B & D & \bar{D} & 1 & \bar{B}_1 & 0 & \bar{D} & \bar{D} \end{array}$$

(Using cube e gives a D-cube with a non-input component that is B_0 .)

The algorithm for completing a D-cube D is thus:

1. Find the right-most component for which a constant value is specified - say this is component k and has value A (= 0 or 1).
2. If this is not an input lead, choose a cube from the singular cover of the gate whose output is lead k and which has a value A for lead k. Combine this with D using the operation of intersection to form a new D-cube D'. Try a different cube from the singular cover for this gate if any components which are not input leads are subscripted B's or D's or if the intersection is empty.
3. Find the next constant lead to the left of k in D', say this is k'. Set k = k', D = D', and repeat step 2. If all constants have been processed, the resulting D-cube D' is complete.

The entire set of completed D-cubes for the network of Figure 1.3 is:

	1	2	3	4	5	6	7	8	9	10	11
	0	D	1	D	B	x	0	1	D	0	D
	0	D	0	B	D	\bar{D}	1	1	0	\bar{D}	\bar{D}
D_1	0	1	D	B	x	x	\bar{B}_0	D	0	D	
D	0	1	D	B	x	0	x	D	0	D	
D_0	0	0	B	D	\bar{D}	1	\bar{B}_1	0	\bar{D}	\bar{D}	
1	x	D	1	1	0	x	0	D	0	D	
x	1	D	1	1	0	x	x	D	0	D	
0	0	x	0	0	1	D	1	0	D	D	
\bar{B}_0	0	x	0	0	1	1	D_1	0	D	D	

1.5 FORMATION OF TESTS FROM COMPLETED D-CUBES

The final step in finding possible test vertices for a network is to form T-cubes of tests from the completed D-cubes of its sensitized paths. This is done in two steps. First, "primitive T-cubes" are

formed directly from the D-cubes of the sensitized paths. Then primitive T-cubes are combined to form "composite T-cubes" of tests which sensitize several paths simultaneously. These steps are described below.

1.5.1 Formation of Primitive T-Cubes from Completed D-Cubes

From each completed D-cube of a sensitized path ($\neq \phi$) either one or two possible tests (depending on whether it contains D_0 , D_1 , etc.) can be formed since, when a path is sensitized, a change in the value of a lead on that path will cause a corresponding change in the output. The T-cubes formed from these D-cubes are called primitive T-cubes since the tests they represent may sensitize paths in the network other than that of the D-cube it was derived from, and therefore may actually test more leads than are indicated in the primitive T-cube. (E.g., note that if lead 6 is 1, T-cube T_b shown below corresponds to an input vertex which also tests leads 7 and 10 for s-a-1, i.e., it may also sensitize the path 7-10-11.) Using the T-cube notation for tests presented earlier, it can be seen that from the D-cube

1	2	3	4	5	6	7	8	9	10	11
0	D	1	D	B	x	0	1	D	0	B

two primitive T-cubes can be formed:

	1	2	3	4	5	6	7	8	9	10	11
$T_a =$	0	T	1	T	1	x	0	1	T	0	T
$T_b =$	0	\bar{T}	1	\bar{T}	0	x	0	1	\bar{T}	0	\bar{T}

since setting input lead 2 to 1 allows any s-a-0 fault along the path to be detected, and setting input lead 2 to 0 allows any s-a-1 fault along the path to be detected.

On the other hand, from the D-cube

1	2	3	4	5	6	7	8	9	10	11
D_0	0	0	B	D	\bar{D}	1	B_1	0	\bar{D}	\bar{D}

only one primitive T-cube can be formed:

1	2	3	4	5	6	7	8	9	10	11
\bar{T}	0	0	0	\bar{T}	T	1	1	0	T	T

since the appearance of D_0 indicates that the path can be sensitized only when input lead 1 is 0. (Note that B's are changed to 0's or 1's since they are not part of the sensitized path.)

Thus, to form primitive T-cubes from a D-cube one must:

1. If the D-cube has no components which are D_0 , B_0 , \bar{D}_1 , or \bar{B}_1 ; form a T-cube by modifying the non-constant components of the D-cube as follows:

$D \rightarrow T$	$B \rightarrow 1$
$\bar{D} \rightarrow \bar{T}$	$\bar{B} \rightarrow 0$
$D_1 \rightarrow T$	$B_1 \rightarrow 1$
$\bar{D}_0 \rightarrow \bar{T}$	$\bar{B}_0 \rightarrow 0$

(Here we are setting $D = 1$; so that leads on the path marked with D (\bar{D}) will detect s-a-0 (s-a-1) faults. Thus D (\bar{D}) is replaced with T (\bar{T}). B (\bar{B})'s are changed to 1 (0)'s since they are not part of the sensitized path. Assigning $D = 1$ is consistent with the cube since it contains no D_0 , B_0 , \bar{D}_1 , or \bar{B}_1 components.)

2. If the D-cube has no components which are D_1 , B_1 , \bar{D}_0 , or \bar{B}_0 ; form a T-cube by modifying the non-constant components of the D-cube as follows:

$$\begin{array}{ll}
 D \rightarrow \bar{T} & B \rightarrow 0 \\
 \bar{D} \rightarrow T & \bar{B} \rightarrow 1 \\
 D_0 \rightarrow \bar{T} & B_0 \rightarrow 0 \\
 \bar{D}_1 \rightarrow T & \bar{B}_1 \rightarrow 1
 \end{array}$$

(Here we are setting $D = 0$, so that leads marked with D (\bar{D}) detect s-a-1 (s-a-0) faults.)

The primitive T-cubes for the network of Fig. 1.3 are:

	1	2	3	4	5	6	7	8	9	10	11
a	0	T	1	T	1	x	0	1	T	0	T
b	0	\bar{T}	1	\bar{T}	0	x	0	1	\bar{T}	0	\bar{T}
c	0	T	0	1	T	\bar{T}	1	1	0	\bar{T}	\bar{T}
d	0	\bar{T}	0	0	\bar{T}	T	1	1	0	T	T
e	T	0	1	T	1	x	x	0	T	0	T
f	T	0	1	T	1	x	0	0	T	0	T
g	\bar{T}	0	1	\bar{T}	0	x	0	1	\bar{T}	0	\bar{T}
h	\bar{T}	0	0	0	\bar{T}	T	1	1	0	T	T
i	1	x	T	1	1	0	x	0	T	0	T
j	1	x	\bar{T}	1	1	0	x	0	\bar{T}	0	T
k	x	1	T	1	1	0	x	x	T	0	T
l	x	1	\bar{T}	1	1	0	x	x	\bar{T}	0	\bar{T}
m	0	0	x	0	0	1	T	1	0	T	T
n	0	0	x	0	0	1	\bar{T}	1	0	\bar{T}	\bar{T}
o	0	0	x	0	0	1	1	T	0	T	T

1.5.2 Formation of Composite T-Cubes from Primitive T-Cubes

In order to find tests which sensitize many paths simultaneously, primitive T-cubes must be merged into composite T-cubes. Only compatible T-cubes may be merged. T-cubes are compatible if the lead assignments they imply do not conflict. In merging two T-cubes, we again want to find those vertices which are consistent with both cubes. Thus, it is also an intersection operation, and, as before, can be

performed component by component. If the cubes to be merged are compatible, then the input assignment implied by the new cube detects all faults which either of the given cubes detect. Thus, the following table specifies the rules for component-by-component merging:

	x	0	1	T	\bar{T}
x	x	0	1	T	\bar{T}
0	0	0	ϕ	ϕ	\bar{T}
1	1	ϕ	1	T	ϕ
T	T	ϕ	T	T	ϕ
\bar{T}	\bar{T}	\bar{T}	ϕ	ϕ	\bar{T}

If any ϕ 's result when merging is attempted, then the T-cubes are incompatible.

(It should be noted that only the input variable assignments need be considered in order to determine if two T-cubes formed from complete D-cubes are compatible. This is because complete D-cubes have all necessary assignments specified in terms of input leads. This fact simplifies finding the sets of primitive T-cubes which can be merged. Of course, once the T-cubes are known to be compatible, the component by component merging must be performed to determine the faults it detects.)

Merging of primitive T-cubes will be illustrated on the primitive T-cubes derived from the network of Figure 1.3 shown in the previous section. It can easily be seen that the following groups are compatible: ak, bgn, dhmo, cl, ei, j. Merging these into composite T-cubes gives:

	1	2	3	4	5	6	7	8	9	10	11	<u>T-Cubes Merged</u>
1	0	T	T	T	1	0	0	1	T	0	T	ak
2	\bar{T}	\bar{T}	1	\bar{T}	0	x	\bar{T}	1	\bar{T}	\bar{T}	\bar{T}	bgn
3	0	\bar{T}	0	0	\bar{T}	T	T	T	0	T	T	dhmo
4	0	T	\bar{T}	1	T	\bar{T}	1	1	\bar{T}	\bar{T}	\bar{T}	cl
5	T	0	T	T	1	x	x	0	T	0	T	ei
6	1	x	T	1	1	0	x	0	T	0	T	j

(Cube f is a subcube of cube e.)

The table resulting from forming all maximal composite tests for a network can be treated as a fault table, and the minimum test set can be found from it using well-known methods for solving covering problems. For the example above, the minimum test set consists of T-cubes 2, 3, 4, and 5. (Note that none of the tests test line 8 for s-a-1. This is because this fault is not detectable since the \bar{A} input to the AND gate is redundant.)

1.6 ALGORITHM FOR FINDING AN APPROXIMATELY MINIMAL TEST SET FOR A GIVEN NETWORK

1. Form the singular cover and primitive D-cubes of the network.
2. Form all possible complete D-cubes (containing no subscripted B's as non-input components) of all paths from an input to an output in the network. (A fairly simple recursive algorithm based on the singular cover and primitive D-cube tables shown earlier can be used to do this.) Delete any cubes which are contained in other cubes.
3. Form all possible primitive T-cubes from these D-cubes.
4. Form all possible maximal composite T-cubes from these primitive T-cubes. This can be done by examining only the input assignments

of the primitive T-cubes to find those which are compatible, then merging these.

5. The resulting table can then be treated as a "fault table", and from it one can find the minimum set of tests which cover all faults, using well-known methods of solving covering problems.

1.7 FAULTS WHICH CANNOT BE DETECTED BY SENSITIZING SINGLE PATHS FROM AN INPUT TO AN OUTPUT

So far, we have considered finding test sets only for faults which can be detected by sensitizing single paths from inputs to outputs. Unfortunately, it is not true that all faults can be detected in this manner. Several networks have been found which have faults that cannot be detected by tests of the type derived above. (See Appendix 1.1.)

These networks fall into two classes:

1. A fault (not on an input lead) which can only be detected by sensitizing a single path from it to the output. (No sensitizable path from an input to an output passes through it.)
2. A fault which can only be detected by sensitizing a multiple path - this occurs when networks contain reconverging fanout and some fault before the fanout can only be detected by the changes that it causes on two or more reconverging paths.

The algorithm can be modified to find tests for faults of these types (and tests for all detectable faults can be found using the modified algorithm [2]). The modifications necessary for each case are discussed below.

1.7.1 Class 1

Tests for faults in this category can be found simply by using the algorithm described earlier to find a sensitized single path from the lead to be tested to an output. One must also use the singular cover to find an input assignment which makes the lead to be tested 0 or 1, depending on whether a s-a-1 or s-a-0 fault is to be detected. A T-cube can be formed from the resulting D-cube as before, since the test will also detect faults along the sensitized path.

1.7.2 Class 2; Sensitizing Multiple Paths

Finding tests for faults which can be detected only by multiple sensitized paths is more difficult, and it requires several modifications to the method described above. With these modifications, the algorithm becomes essentially the same as that described by Roth [2,3].

The primary differences between the algorithm previously described and that which must be used when multiple paths are to be sensitized are that:

1. When fanout occurs, one must attempt to make the D's propagate along 2 or more of the resulting paths, and
2. When paths reconverge, a primitive D cube for a gate must be formed which has D's on several input leads.

The first of these creates no problems--when the path is known, one simply chooses a primitive D-cube for each gate on the path which causes the D to propagate from the input lead on the path to the output. For example, consider the network shown in Appendix 1.1.2. Intersecting the following primitive D-cubes

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B			D	0	B	B	1								
		1			D							B			
						D		1					B		

gives the D-cube

$$D_a = B \quad 1 \quad D \quad 0 \quad D \quad D \quad 1 \quad 1 \quad \quad \quad B \quad B$$

Note that leads 13 and 14 now have B's; i.e., if lead 4 is 0 (1), then leads 13 and 14 are both 0 (1). The primitive D-cube of the final OR gate which is needed to complete the path to the output is one which specifies that if leads 13 and 14 are both 0 (1), then lead 16 will be 0 (1). Clearly, this is the D-cube:

$$D_b = \begin{array}{cccc} & \underline{13} & \underline{14} & \underline{15} & \underline{16} \\ & D & D & 0 & D \end{array}$$

Note that this D-cube does not mean that if lead 13 is 0 (1), then leads 14 and/or 16 are 0 (1). Thus, D-cubes with two or more D's on input leads to the same gate have a slightly different interpretation than those used previously. Other D-cubes of this type for the same gate are:

$$\begin{array}{cccc} & \underline{13} & \underline{14} & \underline{15} & \underline{16} \\ D & D & D & D & D \\ 0 & D & D & D & D \\ D & 0 & D & D & D \end{array}$$

Intersecting D_a and D_b shown above to complete the multiple path to the output gives:

$$D_a \cap D_b = \begin{array}{cccccccccccccccc} & \underline{1} & \underline{2} & \underline{3} & \underline{4} & \underline{5} & \underline{6} & \underline{7} & \underline{8} & \underline{9} & \underline{10} & \underline{11} & \underline{12} & \underline{13} & \underline{14} & \underline{15} & \underline{16} \\ B & & 1 & D & 0 & D & D & 1 & 1 & & & & & D & D & 0 & D \end{array}$$

Completing this cube as before gives the complete cube:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$D_c =$	B	1	1	D	0	D	D	1	x	1	1	0	D	D	0	D

Again, it should be emphasized that this cube does not mean that if, for example, lead 6 changes from 0 to 1 (1 to 0), then the output would change from 0 to 1 (1 to 0). Thus, this cube does not necessarily specify tests which will detect s-a-0 or s-a-1 faults on lead 6 (or 7, 13, or 14). It only specifies a test for leads 4 and 16 (or, in general, all leads on the path before the fanout and after the reconvergence of the multiple path).

The above extensions to find the conditions necessary to sensitize a known multiple path seem quite simple. The difficult problem is determining which multiple path must be sensitized in order to detect a given fault. In general, all possible multiple paths must be tried (including those which contain more than 2 paths sensitized simultaneously). This is quite simple in the example above, since the path used is the only possible one. In complicated networks, however, there may be many possibilities, all of which must be tried. Roth [2,3] describes an organized approach to the problem of trying all of these possible paths, and determining as quickly as possible when a path is not sensitizable.

An interesting problem which arises concerning multiple sensitized paths is determining the faults that the test actually detects. As was mentioned earlier, any such test detects single s-a-1 or s-a-0 faults on those leads before the fanout occurs and after the reconvergence. Consider, however, D-cube D_c above. The test resulting when all D's in the cube are 0 can easily be seen to detect s-a-1 faults on

all leads which have a D in the D-cube. (All of the inputs to the OR are 0, so each of the paths is actually sensitized separately). On the other hand, when all D's are 1, the test does not detect s-a-0 faults on leads 6, 7, 13, or 14. For simple cases such as this, it is easy to see when a test with reconverging paths detects faults on all leads in the paths (if the point of reconvergence is an AND or NAND gate, all inputs must be 1; if it is OR or NOR, all inputs must be 0). The situation is slightly more complicated when several diverging and reconverging paths are simultaneously sensitized, but the same type of reasoning can be used to determine what leads are tested.

Tests which sensitize multiple reconverging paths and detect all faults on the paths may be desirable ones to include in a test set. Unfortunately, trying all multiple paths, in addition to all single paths, does not seem practical at present. Work is presently being done on determining which multiple reconverging paths detect many faults (e.g., if the difference between the number of inversions on two branches of a reconverging path is odd, it can easily be seen that sensitizing this multiple path will never detect all faults on the path).

1.7.3 The Modified Algorithm

The following steps thus must be added to the algorithm of Section 1.6 for the case when the table resulting from step 3 does not include primitive T-cubes containing tests for all faults:

4. Attempt to form a single sensitized path from each fault not detected to an output. If a complete D-cube can be found, form the appropriate primitive T-cubes and add them to the list of T-cubes.

5. Form all possible maximal composite T-cubes from these primitive T-cubes.
6. If there are still faults which are not detected by some test in this table, find a multiple path which can be sensitized from each of these to an output (using the algorithm described in Roth). If no multiple sensitizable path can be found, the fault is not detectable. For each fault detected in this manner, add a T-cube with a T or \bar{T} only on the lead corresponding to the fault.
7. The resulting table can be treated as a fault table and can be solved using standard covering techniques.

1.8 CONCLUSIONS AND DISCUSSION

The method described above for finding approximately minimal test sets for single s-a-0 and s-a-1 faults in arbitrary combinational networks appears to give "good" test sets for the examples tried. Whether or not these test sets are always minimum is not known as yet for networks with fan-out. (As was seen earlier, for networks with no fanout, one need only worry about testing each input, since all other faults are detected by a set of tests which detects all input faults. The algorithm above can be seen to find minimum test sets in this case.) No real attempt has been made to find counterexamples because of the difficulty of determining what the minimum test set actually is.

The method is well suited for computer implementation, since it was developed with computer implementation in mind. Its effectiveness cannot really be judged until it has been programmed and compared with other known methods, both in respect to efficiency and size of test

sets resulting. It is important to note that none of the tables grow exponentially with the number of input variables, gates in network, etc. However, networks with much fanout and therefore many paths from inputs to outputs may require large tables of D-cubes of sensitizable paths, and the resulting fault table may still be large. The method as described gives minimal test sets (if the fault table is solved exactly) within the restriction on the type of tests it finds (because the table of composite T-cubes contains all tests which sensitize single paths from inputs to outputs and which are not "covered" by some other test).

To be practical for large networks, the method should be modified to use some heuristics for finding "good" paths and "good" combinations of primitive T-cubes, and also approximate methods could be used to solve the resulting fault table. Some progress has already been made in this direction, but further development is required.

As was seen in Section 1.7, considerably more computation may be required when tests for some faults cannot be found by sensitizing a single path from the fault to an output. Test sets for nearly all networks which have been tried to date by hand can be found without using the methods of Section 1.7. Whether it is actually true that test sets for "most" networks can be found by considering only single paths should be investigated further only after a programmed version of the algorithm is available.

It should also be pointed out that, although the method can be used to find test sets for networks with undetectable faults (the test set will, of course, detect only the detectable faults), the single-

fault assumption makes these test sets of questionable value. This is because two or more faults must occur before some malfunction can be detected, and the test sets found with a single fault assumption will not necessarily detect this malfunction. In other words, the single-fault assumption is likely to be valid only when all single faults can be detected, and the network is tested often enough so that the probability of multiple faults is small.

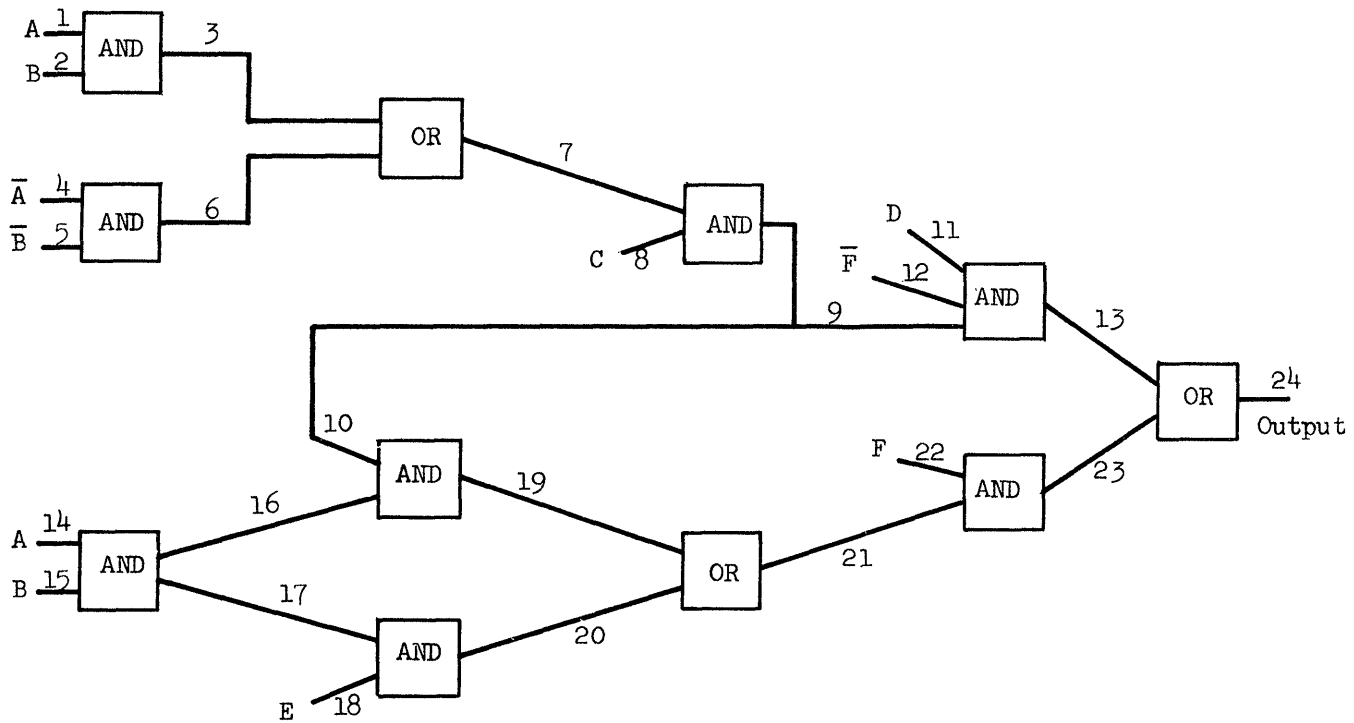
REFERENCES

1. Armstrong, D. B., "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Sets". IEEE Transactions on Elect. Comp., Vol. EC-15, No. 1, Feb. 1966, p. 66.
2. Roth, J. P., "Diagnosis of Automata Failures, A Calculus and a Method", IBM Journal, July 1966, p. 278.
3. Roth, Bouricius, and Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits". IEEE Transactions on Elect. Comp., Vol. EC-16, No. 5, Oct. 1967, p. 567.

APPENDIX 1.1

NETWORKS WITH FAULTS FOR WHICH TESTS CANNOT
BE FOUND BY THE METHOD DESCRIBED

Case 1: A fault, not on an input lead, that can only be detected by sensitizing a path from it to the output (no sensitizable path from an input to an output passes through it).

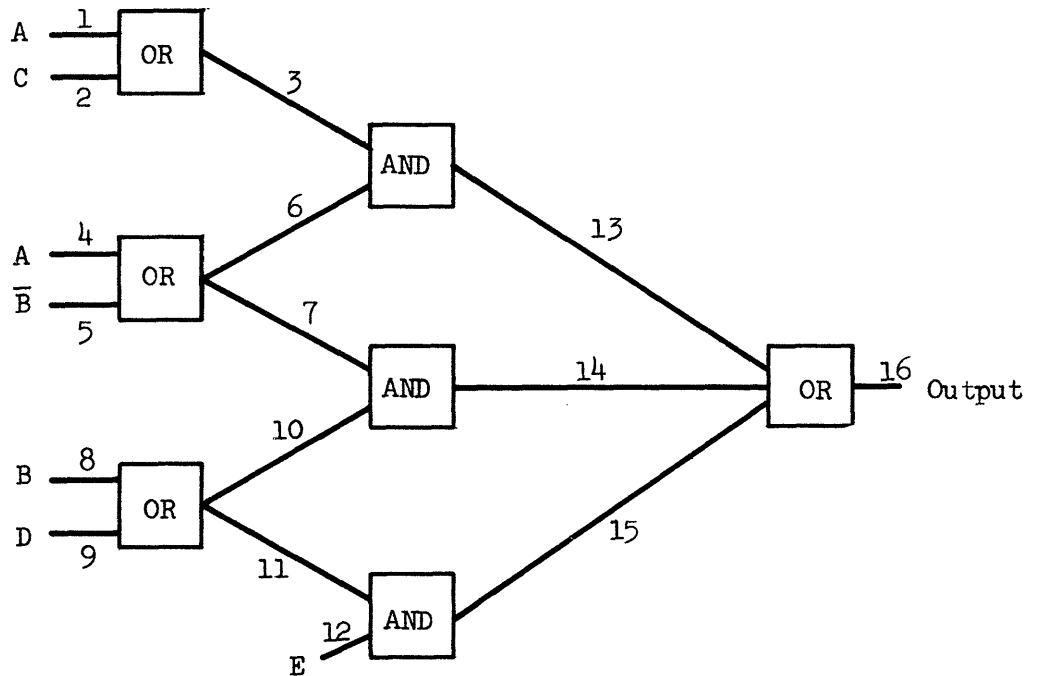


Analyzing this network using the ideas of sensitized paths shows that lead 16 can be tested for s-a-1 only by applying an input vertex with $A = 0$ and $B = 0$. (Since lead 10 must be 1 for the path from lead 16 to the output to be sensitized.) But in this case, there is no path sensitized through the AND gate of which 16 is the output. One can verify that it is impossible to complete the D-cubes corresponding

to either of the paths 14-16-19-21-23-24 or 15-16-19-21-23-24 (the only paths passing through lead 16) so that lead 16 can be tested for s-a-1. However, the following D-cube does contain a test for lead 16 s-a-1, but would not be found by testing only paths from inputs to outputs:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0	0	0	1	1	1	1	1	1	1	x	0	0	0	0	D ₀	0	x	D	0	D	1	D	D

Case 2: A fault which can only be detected by sensitizing a multiple path.



For this network, it can be verified that there is no complete D-cube of a sensitized path which contains a test for lead 4 s-a-0. Note that setting $A = 1$, $\bar{B} = 0$ makes leads 3, 6, 7 and 10 have value 1; thus both leads 13 and 14 have value 1, and, according to our previous observations, the path is not sensitized through the final OR

gate. However, if $E = 0$, and lead 4 becomes 0 due to a fault, both leads 6 and 7 must become 0, and the output does in fact change from 1 to 0. Thus, the input vertex $A = 1, B = 1, C = 0$ or $1, D = 0$ or $1, E = 0$ is a test for lead 4 s-a-0, but a multiple path is sensitized in the sense that a change on lead 4 causes changes in both the paths 4-6-13-16 and 4-7-14-16; and both of the changes are necessary to cause the output to change when the fault occurs. (Note that this vertex does not detect faults on leads 6, 7, 13, or 14.)

Note: It can be verified that there are no redundant leads in either of the above networks; i.e., all leads in both networks can be tested for s-a-0 and s-a-1.

CHAPTER 2

CHECKING OF SINGLE THRESHOLD ELEMENTS

This chapter is concerned with a method for finding a short test to verify, by means of output observations only, that a given threshold gate does in fact realize a specified single-threshold function. It turns out the method is independent of the weights and threshold that were actually used in designing the gate, and it is applicable no matter how many of the weights and/or the threshold are faulty. Furthermore, tight bounds are given on the length of a checking test for all single-threshold functions.

2.1 THE FAILURE MODEL

A failure is present in a logic element if and only if G , the function realized by the element and F , the nominal function, differ for at least one input combination. Clearly the potential presence of any arbitrary type of failure implies that all input combinations must be tested, because any one alone may result in an incorrect output. Therefore, it is assumed here that a failed threshold element (TE) realizes a function G which is STE (single threshold element) realizable. In other words, even a failed element is assumed to be describable by some weight-threshold vector, although incorrect. This assumption is very reasonable from the practical point of view.

Also, we shall assume that the contribution to the weighted sum (WS) of a variable x_k when $x_k = 1$ is constant and independent of the other input values.

Without loss of generality, we shall assume that operation is always according to the "perfect zero" model. This model assumes that when a logical 0 is applied, the actual contribution of that input to the WS is 0. To see that this assumption is indeed not restrictive, suppose that due to noise the contribution to the WS when $x_k = 0$ is $w_{k0} \neq 0$ and when $x_k = 1$ the contribution is w_{k1} . Also assume that the threshold is T. We can then write the WS for any vertex \vec{v} as

$$\sum_{\vec{v}} (w_{k0} + (w_{k1} - w_{k0}) v_i)$$

where $v_i = 1$ (0) if $x_i = 1$ (0) in the vertex \vec{v} . Now let $w_{k1} - w_{k0} = w_k$. Then the WS becomes

$$\sum_{k=1}^n w_{k0} + \sum w_k v_k$$

so that

$$F = 1 \text{ iff } \sum_1^n w_{k0} + \sum w_k v_k \geq T$$

$$F = 0 \text{ iff } \sum_1^n w_{k0} + \sum w_k v_k < T$$

But this is equivalent to

$$F = 1 \text{ iff } \sum w_k v_k \geq T'$$

$$F = 0 \text{ iff } \sum w_k v_k < T'$$

where $T' = T - \sum_1^n w_{k0}$ is the effective threshold when there is excitation due to a logical 0. Hence by using T' we are free to assume the perfect zero model without loss of generality.

2.2 UNATENESS AND POSITIVE WEIGHTS

As is well known, the MSP (minimum sum of products) expression for an STE realizable function is unique and unate, which means that the MSP expression contains each variable either in complemented (for negative unateness) or in uncomplemented form (for positive unateness), but not in both. For the sake of simplicity and without loss of generality, we shall assume that the MSP form is written in those variables which lead to positive weights, i.e., in those variables that lead to positive unateness. As is also well known, every STE realizable function can be constructed with positive weights only. (If the MSP expression contains the variable X in the form \bar{X} , then F can be realized with $w_{\bar{X}} > 0$ if the variable \bar{X} is connected to the gate.) If it happens that the function F on hand is realized with negative weights, then the algorithm in Section 2.6 can be used to quickly find the set of tests for F , given the test which checks the related function H when all weights are positive.

Later on, we shall have need to use the MSP expression of the complement of the function. Since we shall always write functions in terms of the variable forms that lead to positive unateness and therefore positive weights, it follows that the variables in the MSP expression of the complement of the function will have all variables in complemented form.

2.3 PRELIMINARIES

We start with two important definitions. Let P be a prime implicant of F (written in positively unate form). Then to P there corresponds a PI vertex of F , v_t , specified as follows: put a 1 in place of those variables that appear in P , and in the remaining variable positions a 0. Let Q be a prime implicant of \bar{F} (written in negatively unate form). Then to Q there corresponds a PI vertex of \bar{F} , v_f , specified as follows: put a 0 in place of those variables that appear in Q , and in the remaining variable positions a 1.

Example: For $F = A + BC$, the PI vertices of F are $v_t^1(A,B,C) = 100$ and $v_t^2(A,B,C) = 011$; and since $\bar{F} = \bar{A}\bar{B} + \bar{A}\bar{C}$, the PI vertices of \bar{F} are $v_f^1(A,B,C) = 001$ and $v_f^2(A,B,C) = 010$.

Lemma 2.1: If any constant in a PI vertex of F (\bar{F}) is changed from 1 (0) to 0 (1), the resulting vertex is in the false (true) body of F .

Proof: Let F be a function of x_1, x_2, \dots, x_n and let $v_t = 111\dots100\dots0$, corresponding to prime implicant $P = x_1x_2\dots x_j$. Consider $v = 011\dots100\dots0$ which differs from v_t in only the first position. If v is in the true body, then it must be covered by a prime implicant of F . But the only prime implicant in unbarred variables x_1 through x_n which can cover v consists of a subset of the variables x_2 through x_j . But then P cannot be a prime implicant. Therefore v must be in the false body of F . The dual statement is similarly proved.

Lemma 2.2: If the PI vertices of F and \bar{F} are applied to threshold element E , then the proper response of E at all of these vertices assures that all the weights in E are positive.

Proof: (By induction) First, consider the smallest weight, w_1 . There exists a PI vertex of F , $v_t = 1S$, where S is some string of 1's and 0's in positions 2 through n . (If no such v_t existed, F would be independent of x_1 .) By Lemma 2.1, the vertex $v = 0S$ must be in the false body. Furthermore, v is a PI vertex of \bar{F} , for the following reason. If v were not a PI vertex of \bar{F} , then there must be a PI, call it Q , other than that formed by the product of \bar{x}_1 and the variables corresponding to those positions in S where there are zeroes. Say S has 0 in positions i, j, \dots, k, ℓ . Then Q is a product of $\bar{x}_1, \bar{x}_i, \bar{x}_j, \dots, \bar{x}_k, \bar{x}_\ell$ from which one or more literals, say \bar{x}_j through \bar{x}_k , have been deleted. Then the PI vertex of Q , call it v' , has a 1 in positions j through k . But then, because no weight is smaller than w_1 , v' is in the true body, and hence Q cannot be a PI of \bar{F} . Therefore, $0S$ must be a PI vertex of \bar{F} . Proper response to $1S$ and $0S$ then assures that w_1 is positive in E . The same argument holds for every weight that can be set equal to w_1 .

Assume that for all $w_i < w_j$ the test has revealed that $w_i > 0$. Consider a PI vertex of F which has 1 in position j . (Such a vertex must exist, for otherwise F is independent of x_j .) Say $v_t = t1S$, where S is a string of ones and zeroes in variables with weights no less than w_j , and t is a string of ones and zeroes in positions with weights less than w_j . Vertex $v = t0S$ must be in the false body of F

(again by Lemma 2.1). Now v must be covered by a PI of \bar{F} , which has the PI vertex $v_f = uOS$, where u has 1's wherever t has 1's, but u may also have ones where t has zeroes. String S in v_f is identical to that in v_t , for otherwise v_f would be in the true body because each weight to the right of position j is no less than w_j . Now applying v_t and v_f and getting the proper response shows that

$$\sum_{x_i=1 \text{ in } t} w_i + w_j + \sum_{x_i=1 \text{ in } S} w_i \geq T_E$$

and

$$\sum_{x_i=1 \text{ in } t} w_i + \sum_{\substack{x_i=1 \text{ in } u \\ \text{but not in } t}} w_i + \sum_{x_i=1 \text{ in } S} w_i < T_E$$

Together, these two show that

$$w_j > \sum_{\substack{x_i=1 \text{ in } u \\ \text{but not in } t}} w_i$$

By inductive hypothesis, each of the w_i in the summation is greater than zero. Hence $w_j > 0$, as was to be shown.

Theorem 2.1: A sufficient test to assure that element E realizes F consists of applying to E all the PI vertices of F and all the PI vertices of \bar{F} .

Proof: If the response of E is correct at all of the PI vertices, then not only does this check the operation of E at these vertices but, by Lemma 2.2, it also shows that all the weights are positive. This assures that operation is also correct at all the vertices covered by all the prime implicants (of both F and \bar{F}),

since a PI vertex of F gives the lowest excitation due to all the true vertices covered by that PI, and a PI vertex of \bar{F} gives the highest excitation due to all the false vertices covered by that PI. But all the true (false) vertices of F are covered by the set of all prime implicants of F (\bar{F}). Hence the test prescribed in the theorem assures correct operation at all vertices.

Having found the sufficient test set, we will now derive the specification of the test set which is necessary. First, we introduce the two terms MTV and MFV. Given a weight-threshold vector \vec{w}, T that realizes F . An MTV (MFV) of that realization is a true (false) vertex of F at which the excitation is no higher (lower) than at any other true (false) vertex. In other words, an MTV is a minimum true vertex and an MFV is a maximum false vertex.

Theorem 2.2: A checking sequence for an element that supposedly realizes F must contain all the MTV's and all the MFV's of every weight-threshold vector that realizes F .

Proof: As observed in the proof of Theorem 2.1, a PI vertex of F (\bar{F}) has excitation lower (higher) than that of any other true (false) vertex covered by that PI. Therefore, an MTV (MFV) is always a PI vertex of F (\bar{F}). Now we shall show that given that all but one MTV or MFV of F and \bar{F} have been checked, one can still have incorrect operation at the omitted vertex. The proof will be constructive.

Let \vec{w}, T denote a weight-threshold vector that realizes F and \vec{w}', T' a second weight-threshold vector related to \vec{w}, T in the following manner. Let v be an MTV of \vec{w}, T with 1 in positions 1 through m and 0

elsewhere. Then let $w'_i = w_i$ for i not in the interval 1 through m , and $w'_i = w_i - u$ for i in the interval 1 through m . Now let $T' = T + d$. In the above, u and d are positive constants, so that surely the function realized by \vec{w}', T' will be 0 wherever $F = 0$. It is possible to select u and d such that \vec{w}', T' realizes a function that is false at v yet true at every other PI vertex of F . This can be seen as follows. Since v is a PI vertex, the ones in positions 1 through m of v cannot all be ones of some other PI vertex of F , so that every other PI vertex has at most $m-1$ ones in positions 1 through m . With \vec{w}' related to \vec{w} as specified, it follows that at v the WS due to \vec{w}' differs from that due to \vec{w} by mu , i.e., $WS'(v) = WS(v) - mu$, whereas at any other PI vertex V of F , $WS'(V) \geq WS(V) - (m-1)u$. Consequently, the construction has resulted in an excitation at v which is strictly less than that at any other PI vertex of F , because v is an MTV of F . Now d can be so chosen that T' lies between $WS'(v)$ and the smallest value of $WS'(V)$. Then testing operation at all the PI vertices of F except v will reveal operation according to F , and yet operation at v is not according to F . Therefore, v must be in the test sequence, since the particular element on hand may be that described by \vec{w}', T' .

A similar argument can be made about the MFV's by letting u and d be negative.

Theorem 2.3: The smallest set of checking tests consists of PI vertices only.

Proof: (By contradiction) Suppose there exists a smallest set of checking tests, α , in which at least one true vertex V is not a

PI vertex of F . Let V_p be the PI vertex of a PI that covers V . Then if the excitation due to V_p is denoted by $WS(V_p) = s_a$, then $WS(V) = s_a + s_b$, where s_b is the sum of those weights that contribute to $WS(V)$ and not to $WS(V_p)$. If V is applied, then it verifies that $s_a + s_b > T$. If this inequality could be used to verify that operation at V_p is correct, then in α there would have to exist a subset β which jointly specifies that $s_b < 0$. But since the function is positively unate, no such set β can exist. Therefore, correct operation at V_p must be established by some set γ of tests, and this set cannot contain V . Since γ verifies operation at V_p , the only information that applying V could contribute is that $s_b > 0$. But then there would have to exist a subset δ of the tests which shows that $s_a < T$. Set δ cannot exist, since in fact $s_a > T$. Consequently, vertex V contributes no information, and therefore it cannot be in the smallest test set.

The proof for false vertices that are not PI vertices of \bar{F} is similar.

2.4 ALGORITHM FOR FINDING SHORT TEST SETS

Having established in Theorem 2.1 that a sufficient test consists of the set P of all the PI vertices, in Theorem 2.2 that the necessary test set is a subset of P , and in Theorem 2.3 that the shortest test consists only of elements in P , we are now faced with the task of pruning from P those elements that are not necessary, so that an efficient test will result.

In fact, in most functions not all of the PI vertices need be applied. For example, consider the function

$$F = DE + BE + ACD + \dots$$

$$\bar{F} = \bar{D}\bar{E} + \dots$$

In a gate allegedly realizing F , proper response to vertex 10110 (the PI vertex of ACD) shows that

$$w_A + w_C + w_D \geq T$$

and proper response to vertex 11100 (the PI vertex of $\bar{D}\bar{E}$) shows that

$$w_A + w_B + w_C < T.$$

Together these two relations imply that

$$w_D > w_B$$

Consequently, if the gate also responds correctly to vertex 01001 (i.e., if $w_B + w_E \geq T$), then one can also be sure that the gate will respond properly to vertex 00011, which is the PI vertex of DE . Therefore, this vertex can be eliminated from the test set.

In general, a vertex v can be eliminated and will be called redundant whenever correct operation at v can be deduced from correct operation at some of the vertices actually tested.

Note that if a true (false) vertex v is redundant, then it can never be an MTV (MFV) in any realization. In other words, a necessary condition for redundancy of a true (false) vertex v is that in every realization there exists a true (false) vertex v' such that $WS(v) > WS(v')$ ($WS(v) < WS(v')$). Conversely, if this inequality has been verified by some test not involving v , then v need not be

included in the test, if the test also shows that the response to v' is true (false).

Henceforth, let v and v' be true vertices. If $WS(v) > WS(v')$ in every realization, then this ordering must be specified by the function. Now the function specifies excitations as sums of weights greater or equal (smaller) than the threshold for true (false) vertices. A true vertex v_t together with a false vertex v_f specifies that the sum of the weights contributing to $WS(v_t)$ is greater than the sum of the weights contributing to $WS(v_f)$. Similarly, pairs of vertices, where in each pair one vertex is true and the other vertex is false, show that a sum of weights is greater than some other sum of weights. If $WS(v) > WS(v')$ in every realization, then this ordering must be revealed by some pairs of vertices v_{tj}, v_{fj} . These ideas are illustrated in the previously given example, where $v_t = 10110$, $v_f = 11100$, $v' = 01001$, and $v = 00011$. Here a single pair (v_t and v_f) is sufficient to show that in all realizations $WS(v) > WS(v')$. In general, however, it is not true that a single pair v_t, v_f is sufficient to show that $WS(v) > WS(v')$.

This discussion should explain why our strategy for finding redundant PI vertices will be as follows. We compare the excitation at each PI vertex of F in P , call it v , with that at every other PI vertex of F , call it v' . If there exist a v_t and a v_f in P which show that $WS(v) > WS(v')$, then we eliminate v from the test set. Similarly, we compare the excitation at each PI vertex of \bar{F} in P , call it v , with that at every other PI vertex of \bar{F} , call it v' . If

there exist v_t and v_f in P which show that $WS(v) < WS(v')$, then we eliminate v from the test set.

The procedure is given more precisely in the algorithm below, where we refer to four sets of PI vertices: T, T', C, C' . Set T (C) consists of all the PI vertices of F (\bar{F}); its elements are denoted by t_i (c_i). Set T' (C') is a subset of T (C).

ALGORITHM

1. List all the t_i and all the c_i .
2. Set $T' = T$ and $C' = C$.
3. Pick an ordered pair $t_i; t_j$ and compute the difference vector, \vec{d}_{ij} , as follows. Vector \vec{d}_{ij} has as its components all those variables in uncomplemented form which appear in t_i and not in t_j . Also, \vec{d}_{ij} has as its components in complemented form those variables which appear in t_j and not in t_i . All those variables which appear in both t_i and t_j do not appear in \vec{d}_{ij} .
4. Find (through exhaustion) a third term c_k in C such that (a) the complemented variables appearing in \vec{d}_{ij} do not appear in c_k and (b) the uncomplemented variables in \vec{d}_{ij} do appear in c_k .
5. Find (through exhaustion) a fourth term t_h in T which does not contain (a) all variables which appear in \vec{d}_{ij} complemented; and (b) all variables which appear in c_k and do not appear in \vec{d}_{ij} uncomplemented.

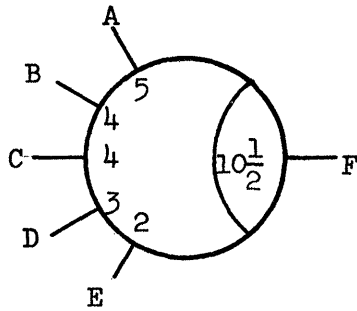
6. If c_k and t_h exist, eliminate t_i from T' . If not, search for another c_k until either a c_k, t_h pair is found or all the possibilities for c_k have been exhausted.
7. Return to Step 3 and consider another pair t_m, t_n until all ordered pairs have been tried.
8. Replace c for t , C for T , C' for T' and vice-versa and repeat Steps 3 through 7.

The execution of the algorithm is illustrated in Figure 2.1.

Let us examine the steps in the algorithm to understand what they actually do. Step 3) selects an ordered pair of vertices (or terms) and computes the difference vector. The difference vector is shorthand not only for the way vertices \vec{v}_i and \vec{v}_j differ, but for the ordering of the sums of weights which would indeed order vertices \vec{v}_i and \vec{v}_j . For example, suppose $\vec{d}_{ij} = abc\bar{d}\bar{e}\bar{f}$. Then this is a shorthand for $w_a + w_b + w_c > w_d + w_e + w_f$ and if this inequality can be shown to hold, then $WS(\vec{v}_i) > WS(\vec{v}_j)$, and \vec{v}_i need not be tested. The purpose of c_k and t_h is to verify this inequality.

We will refer to the sum $w_a + w_b + w_c$ as the "left sum" and the sum $w_d + w_e + w_f$ as the "right sum". Since all weights are positive, if the sum of the weights forming the left sum can be shown to be greater than a sum containing at least all of the weights making up the right sum, the ordering required by the difference vector will be established and $WS(\vec{v}_i)$ will be strictly greater $WS(\vec{v}_j)$. Vertex \vec{v}_i can then be eliminated from the test set. In other words, for

$$w_a + w_b + w_c > w_d + w_e + w_f$$



ABC DE	000	001	011	010	110	111	101	100
00	0	0	0	0	0	1	0	0
01	0	0	0	0	1	1	1	0
11	0	0	1	0	1	1	1	0
10	0	0	1	0	1	1	1	0

$$F = ABC + AB D + AB E + A CD + A C E + BCD$$

11100 11010 11001 10110 10101 01110 True MSP Vertices

$$\bar{F} = \bar{A}\bar{B} + \bar{A}\bar{C} + \bar{A}\bar{D} + \bar{B}\bar{C} + \bar{C}\bar{D}\bar{E} + \bar{B}\bar{D}\bar{E}$$

00111 01011 01101 10011 11000 10100 False MSP Vertices

t_i	t_j	\bar{d}_{ij}	c_k	not in t_h	t_h	Comment
ABC	ABD	$\bar{C}\bar{D}$	$\bar{A}\bar{C}$ $\bar{B}\bar{C}$	A,D B,D	ACE	Eliminate ABC
ABD	ABE	$\bar{D}\bar{E}$	$\bar{A}\bar{D}$	A,E	BCD	Eliminate ABD
ABE	ACD	$\bar{B}\bar{C}\bar{D}\bar{E}$	$\bar{A}\bar{B}$ $\bar{B}\bar{D}\bar{E}$	A,C C,D,E		
	ACE	$\bar{B}\bar{C}$				
	BCD	$\bar{A}\bar{C}\bar{D}\bar{E}$				11001 in test set
ACD	ABE	$\bar{B}\bar{C}\bar{D}\bar{E}$				
	ACE	$\bar{D}\bar{E}$	$\bar{A}\bar{D}$	A,E	BCD	Eliminate ACD
ACE	ABC	$\bar{B}\bar{E}$	$\bar{C}\bar{D}\bar{E}$	B,C,D		
	ABD	$\bar{B}\bar{C}\bar{D}\bar{E}$				
	ABE	$\bar{B}\bar{C}$	$\bar{A}\bar{C}$ $\bar{C}\bar{D}\bar{E}$	A,B B,D,E		
	BCD	$\bar{A}\bar{B}\bar{D}\bar{E}$				10101 in test set
BCD	ABC	$\bar{A}\bar{D}$	$\bar{C}\bar{D}\bar{E}$ $\bar{B}\bar{D}\bar{E}$	A,C,E A,B,E		
	ABD	$\bar{A}\bar{C}$	$\bar{B}\bar{C}$ $\bar{C}\bar{D}\bar{E}$	A,B A,D,E		
	ABE	$\bar{A}\bar{C}\bar{D}\bar{E}$				
	ACD	$\bar{A}\bar{B}$	$\bar{B}\bar{D}\bar{E}$ $\bar{B}\bar{C}$	A,C A,C		
	ACD	$\bar{A}\bar{B}\bar{D}\bar{E}$				01110 in test set

Note that since $ABC > ABD > ABE$, ABE need only be compared to three t_j

Figure 2.1
Illustration of
Checking Set
Reduction Algorithm

c_i	c_j	\vec{d}_{ij}	t_k	not in c_h	c_h	Comment
$\bar{A}\bar{B}$	$\bar{A}\bar{C}$	$\bar{B}\bar{C}$	ABD	A,D,C		
			ABE	A,E,C		
	$\bar{A}\bar{D}$	$\bar{B}\bar{D}$	ABC	A,C,D		
			ABE	A,D	$\bar{B}\bar{C}$	Eliminate $\bar{A}\bar{B}$
$\bar{A}\bar{C}$	$\bar{A}\bar{D}$	$\bar{C}\bar{D}$	ABC	A,B,D		
			ACE	A,D,E	$\bar{B}\bar{C}$	Eliminate $\bar{A}\bar{C}$
$\bar{A}\bar{D}$	$\bar{B}\bar{C}$	$\bar{A}\bar{B}\bar{C}\bar{D}$				
	$\bar{C}\bar{D}\bar{E}$	$\bar{A}\bar{C}\bar{E}$	ABD	B,C,D,E		
	$\bar{B}\bar{D}\bar{E}$	$\bar{A}\bar{B}\bar{E}$	ACD	B,C,D,E		01101 in test set
$\bar{B}\bar{C}$	$\bar{A}\bar{C}$	$\bar{A}\bar{B}$	BCD	A,B,D		
	$\bar{A}\bar{D}$	$\bar{A}\bar{B}\bar{C}\bar{D}$				
	$\bar{C}\bar{D}\bar{E}$	$\bar{B}\bar{D}\bar{E}$	ABC	A,C,D,E		
	$\bar{B}\bar{D}\bar{E}$	$\bar{C}\bar{D}\bar{E}$	ABC	A,B,C,D		10011 in test set
$\bar{C}\bar{D}\bar{E}$	$\bar{A}\bar{C}$	$\bar{A}\bar{D}\bar{E}$				
	$\bar{A}\bar{D}$	$\bar{A}\bar{C}\bar{E}$	ACE	A,C,D		
	$\bar{B}\bar{D}\bar{E}$	$\bar{B}\bar{C}$	ACD	A,B,D		
			ACE	A,C,D		11000 in test set
$\bar{B}\bar{D}\bar{E}$	$\bar{A}\bar{C}$	$\bar{A}\bar{B}\bar{C}\bar{D}\bar{E}$				
	$\bar{A}\bar{D}$	$\bar{A}\bar{B}\bar{E}$	BCD	A,C,D		
	$\bar{B}\bar{C}$	$\bar{C}\bar{D}\bar{E}$	ABD	A,B,C		
			ABE	A,C,D		
	$\bar{C}\bar{D}\bar{E}$	$\bar{B}\bar{C}$	ABD	A,C,D		
			ABE	A,C,E		10100 in test set

Illustration of Checking Set Reduction Algorithm
Figure 2.1 (Continued)

it is sufficient to show that

$$w_a + w_b + w_c > w_d + w_e + w_f + w_g$$

Step 4) searches for a term, c_k , in the set of PI's of \bar{F} in which those variables which are barred in \vec{d} do not appear. This simply assures that all the members of the right sum do contribute to $WS(c_k)$. The second requirement specified in Step 4) is not necessary, but improves the efficiency of the algorithm. If any of the left-sum weights came into play at \vec{c}_k , then there would exist no t_h which could establish, together with c_k , that the hypothesized ordering of the sums holds.

Step 5) of the algorithm looks for a fourth term t_h in the set of PI's of F for which those variables which appear in \vec{d} barred do not appear. This is equivalent to saying that all the weights of the right sum must not come into play at the vertex associated with t_h . Also, t_h must not contain any variables which do appear in c_k and do not appear in \vec{d} unbarred. If t_h meets this condition, then the only weights which come into play at \vec{v}_h and not at \vec{v}_k are those that form the left sum.

Every time a term t_i (or vertex \vec{v}_i) is eliminated from the test set, we see that there are three remaining vertices \vec{v}_j , \vec{v}_h , and \vec{v}_k . Vertices \vec{v}_h and \vec{v}_k establish an ordering which, together with the fact that all the weights are positive, requires that $WS(\vec{v}_i)$ be strictly greater than $WS(\vec{v}_j)$. If the vertices \vec{v}_j , \vec{v}_h , and \vec{v}_k are tested and give the proper output, we can be positive that \vec{v}_i will produce the correct output. Therefore \vec{v}_i may be eliminated.

There are several refinements which can be made to improve the efficiency of the algorithm. First of all, it is clear that, if it has been found that $WS(v_1) > WS(v_2)$ and also that $WS(v_2) > WS(v_3)$, it cannot be true that $WS(v_3) > WS(v_1)$ and therefore there is no point in testing this hypothesis. This is illustrated in the example of Fig. 2.1 where $v_1 = 11100$, $v_2 = 11010$, $v_3 = 11001$, and the unnecessary steps are noted in the margin.

Second, if one is given the nominal realization of the function, he can compute the MTV's and MFV's of that realization and, because of Theorem 2.2, one knows that these PI vertices cannot be eliminated. For the realization given in Fig. 2.1, vertices 01110, 11001, and 10101 are MTV's; vertices 01101 and 10011 are MFV's. Note, however, that the test set also includes vertices 11000 and 10100, which are not MFV's in this realization.

Finally, one can use the weight orderings which are determined from the MSP expression of the function to eliminate vertices. We recall the following well-known theorem.

Theorem 2.4: The MSP expression of an STE realizable function F prescribes the following ordering on the set of weights that realize F :

1. All weights for variables that appear in PI's of length L are greater than the weights for variables that appear only in PI's of length greater than L .
2. If two variables x_i and x_j appear an equal number of times in all the PI's of length L , for every L , then w_i and w_j can be made equal.

3. If two variables x_i and x_j appear an equal number of times in all the PI's of length L or less, but x_i appears more often than x_j in PI's of length $L+1$, then $w_i > w_j$.

To illustrate the use of this theorem, consider the example of Fig. 2.1, where $F = ABC + ABD + ABE + ACD + ACE + BCD$. Here all the PI's are of equal length. We have

Variable	Occurrences
A	5
B	4
C	4
D	3
E	2

Hence $w_A > w_B = w_C > w_D > w_E$. (It happens that here the number of occurrences can be used as a weight assignment, but this is not in general true.) Consequently, we know that it must be true that $WS(11100) > WS(11010) > WS(11001)$ and $WS(10110) > WS(10101)$. It so happens that in this example the weight ordering specified by Theorem 2.4 leads to the elimination of all the t_i that can be eliminated, but in general this is not true.

With the refinements discussed above, the testing algorithm can be executed rapidly. But we were not able to prove that the tests remaining in T' and C' are the minimum test sets. In all examples tried, this was found to be so, and it seems reasonable to conjecture that the algorithm does in fact find the smallest test set.

2.5 BOUNDS ON THE LENGTH OF THE CHECKING SET

It turns out that one can find precise bounds on the length of the checking set for any threshold function. To derive these bounds, we need some preliminary considerations and also the results of Appendix A to this chapter.

Lemma 2.2: The MSP expression of a unate function of n variables and the MSP expression of the complement of that function together have at least $n+1$ prime implicants.

Proof: For $n = 2$, there are two possible unate functions and these have the MSP expressions $F_1 = A+B$ ($\bar{F}_1 = \bar{A}\bar{B}$) and $F_2 = AB$ ($\bar{F}_2 = \bar{A}+\bar{B}$). In either case, there is a total of three prime implicants and so the lemma holds for $n = 2$. Suppose the lemma holds for $n-1$ variables but not for n variables. Then there exists a function G of n variables with a total of n or less prime implicants. Since G is unate, it can be written in the form $G = g+Xh$, where g is the sum of all those prime implicants that do not contain one of the n variables X and Xh is the factored sum of the remaining prime implicants of G . Since $\bar{G} = \bar{g}\bar{X} + \bar{g}h$, the first term on the right is the factored sum of the prime implicants of \bar{G} that contain \bar{X} and the second term is the sum of the remaining prime implicants. Now suppose $h \neq 1, 0$. Then $G|_{X=1} = g+h$ and $\bar{G}|_{X=1} = \bar{g}h$. Now $G|_{X=1}$ can have no more prime implicants than G , and $\bar{G}|_{X=1}$ has at least one less prime implicant than \bar{G} . Functions $G|_{X=1}$ and $\bar{G}|_{X=1}$ are functions of $n-1$ variables. Hence if G and \bar{G} together had n or less prime implicants, then there would exist a function of $n-1$ variables, namely $G|_{X=1}$, which has a total of no more than $n-1$ prime implicants contradicting the hypothesis. To complete the proof, $h = 1$, so that $G = g + X$ and

$G' \Big|_{X=0} = g$, having one less prime implicant than G' , while $\bar{G}' \Big|_{X=0} = \bar{g}$, having the same number of prime implicants as \bar{G}' . Function $G' \Big|_{X=0}$ involves $n-1$ variables and in this case also if G' and \bar{G}' together had n or less prime implicants, then there would exist a function of $n-1$ variables with a total of $n-1$ or less prime implicants. But $h \neq 1, 0$ and $h = 1$ exhaust all the possibilities, since $h = 0$ would imply that G is not a function of n variables. Hence the lemma must be true.

Lemma 2.3: There exists a symmetric unate function F of n variables with a total of $n+1$ prime implicants in the MSP expressions of F and \bar{F} .

Proof: The AND and OR functions satisfy the lemma.

Lemma 2.4: No threshold function F has less than $n+1$ prime implicants in the MSP expressions of F and \bar{F} .

Proof: Direct consequence of Lemmas 2.2 and 2.3.

As is well-known, the AND and OR functions can be realized by a TE with all weights set equal. Therefore, all the PI vertices of these functions can be made MTV's and MFV's so that they must all be tested (see Theorem 2.2). Consequently, these functions require precisely $n+1$ tests for checking correct operation.

Lemma 2.5: There exists no threshold function that requires fewer than $n+1$ tests.

Proof: Each vertex in a test verifies that one inequality of the form $\sum w_i - T \geq 0$ or $\sum w_i - T < 0$ is correct. Clearly enough tests must be executed so that each w_i comes into play at least once, and no two tests can involve the same algebraic sum $\sum w_i - T$. But the smallest number of distinct sums of $n+1$ items (the n weights and the threshold) where each item appears at least once is $n+1$. Therefore, there must be at least $n+1$ tests, as claimed.

By a somewhat involved argument given in Appendix A of this chapter, we show that the symmetric function which is true whenever $\left\lfloor \frac{n+1}{2} \right\rfloor$ or more out of n variables are true has the largest possible number of PI's. Furthermore, since the function is symmetric, it can be realized by setting all weights equal, so that all the PI vertices can be made MTV's and MFV's, and hence they must all be tested.

We are now in a position to state the bounds on the checking-set cardinality in the following theorem.

Theorem 2.5: The number of tests, $T(n)$, required to verify that a threshold gate operates as desired is given by

$$n+1 \leq T(n) \leq \binom{n}{\left\lfloor \frac{n+1}{2} \right\rfloor} + \binom{n}{\left\lfloor \frac{n+1}{2} \right\rfloor - 1} \approx \frac{2^{n+3/2} n^{n+1/2}}{\sqrt{\pi} e^{(n+1)} (n-1)^n}$$

Proof: The lower bound follows from Lemma 2.5 and the discussion preceding it. The upper limit follows from Theorem A.1 and the discussion preceding Theorem 2.5. The approximation is based on Stirling's formula, $k! = k^k e^{-k} \sqrt{2\pi k}$, and is valid for n odd.

2.6 FINDING TEST SEQUENCES FOR RELATED FUNCTIONS

In this section, we wish to show that given an optimum test sequence for an STE realizable function F , we can derive from it an optimum test sequence for all those functions H that are related to F by the characteristic vector. Specifically, the relationship between H and F must be such that the characteristic vector of H , denoted by \vec{b}^H (also called the Chow parameters or first $n+1$ Rademacher Walsh coefficients), can be derived from the characteristic vector of F , \vec{b}^F , by permutation and/or multiplication by -1 of the elements of \vec{b}^F .

The usual method of description of a single threshold element is to specify its weight vector, \vec{w} , and threshold T . The function $F'(\vec{v})$ is specified by

$$F'(\vec{v}) = 1 \text{ iff } \vec{v} \cdot \vec{w} \geq T$$

$$F'(\vec{v}) = 0 \text{ iff } \vec{v} \cdot \vec{w} < T$$

Following Dertouzos¹, we transform F and \vec{v} from $0,1$ notation to $+1,-1$ notation by the following transformations:

$$y_i = 2v_i - 1 \text{ and } F(\vec{v}) = 2F'(\vec{v}') - 1$$

so that

$$v_i = \frac{y_i + 1}{2}$$

and

$$\vec{v} \cdot \vec{w} = \sum_i \frac{w_i}{2} + \sum_i \frac{y_i w_i}{2}$$

Consequently, we can write

$$F = +1 \text{ iff } \sum_i y_i w_i \geq 2T - \sum_i w_i$$

$$F = -1 \text{ iff } \sum_i y_i w_i < 2T - \sum_i w_i$$

Now define the modified weight-threshold (MWT) vector

$\vec{a} = a_1, \dots, a_n; a_0$ as follows:

$$a_i = w_i \text{ for } i > 0 \text{ and } a_0 = -(2T - \sum_i w_i)$$

In terms of the MWT vector and the augmented vector

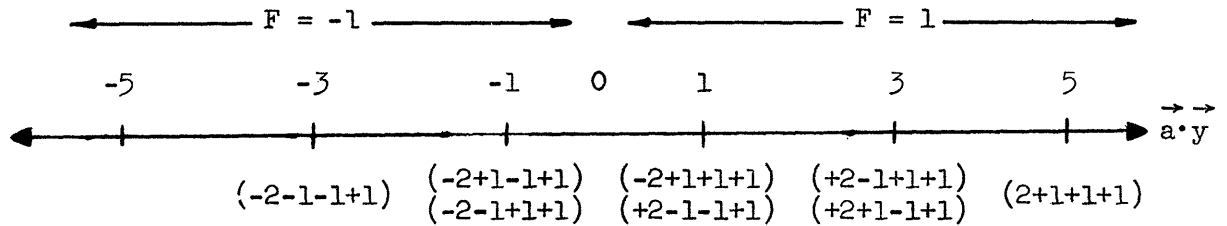
$\vec{y} = y_1, y_2, \dots, y_n; 1$ we have

$$F = +1 \text{ iff } \vec{y} \cdot \vec{a} \geq 0$$

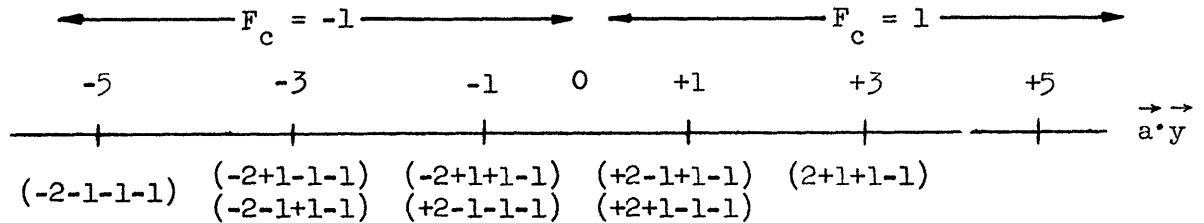
$$F = -1 \text{ otherwise}$$

Each of the 2^n vertices of the n-variable function $F(\vec{y})$ maps onto a point on the excitation line, $\vec{y} \cdot \vec{a}$. For example, let

$\vec{a} = 2, 1, 1, 1$. Then the vertices of $F(\vec{y})$ can be mapped as follows:



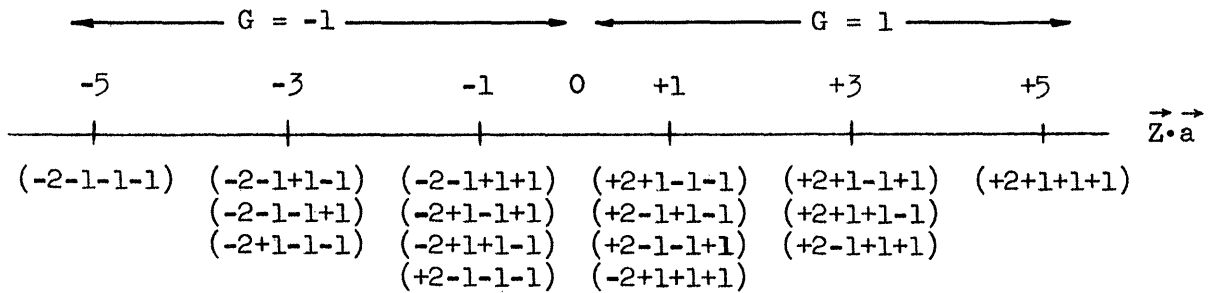
Now let $\vec{a}_c = 2, 1, 1, -1$, realizing $F_c(\vec{y})$. Then we get the plot



Define $\vec{Z} = z_1, z_2, \dots, z_n; z_0$ where each z_i , including z_0 , can take on both the value +1 and the value -1. Also define the function G as follows:

$$G(\vec{Z}) = \begin{cases} 1 & \text{iff } \vec{Z} \cdot \vec{a} \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

The plot of the vertices of $G(\vec{Z})$ for the previous example is



Note that the function G is antisymmetric about 0, i.e., $G(-\vec{Z}) = -G(\vec{Z})$, and the vertices of $G(\vec{Z})$ include all the vertices of F and F_c .

It is easy to see that for any MWT vector it will be true that $G(\vec{Z})$ is antisymmetric and that if at vertex $\vec{Z} = z_1, z_2, \dots, z_0$ the excitation, i.e., $\vec{Z} \cdot \vec{a}$, has value E, then at vertex $\vec{Z}^1 = -z_1, -z_2, \dots, -z_0$ the excitation has value -E.

Consider a function H with characteristic vector \vec{b}^H that differs from the characteristic vector \vec{b}^F of function F only in permutation of the elements and/or multiplication of elements by -1. As Dertouzos has shown, if F can be realized by \vec{a}^F , then H can be realized by \vec{a}^H , where \vec{a}^H differs from \vec{a}^F in exactly the same permutations and/or multiplications by which \vec{b}^H differs from \vec{b}^F . For example, let $F = ABC + ABD + ABE + ACD + ACE + BCD$, which has

$\vec{b}^F = 16, 12, 12, 8, 4; -8$ and is realizable by $\vec{a}^F = 5, 4, 4, 3, 2; -3$.
 The function $H = ad + ae + bc + bde + cd$ has $\vec{b}^H = 16, 8, 12, 8, 4; 12$.
 Therefore $\vec{a}^H = 5, 3, 4, 3, 2; 4$ is a realization of H .

Note for future reference that functions F and H , related as specified above, lead to the same plot of $G(\vec{Z})$ when realized by the related MWT vectors.

A checking sequence for F verifies that the actual weight-threshold vector, $\vec{w};T$ realizes F , and hence it verifies that the actual \vec{a}^F realizes F . But if \vec{a}^F realizes F , then it must also realize $G(\vec{Z})$. This is so for the following reasons. The plot of $G(\vec{Z})$ on the excitation line contains all the excitation values of F , and in addition those where $y_0 = -1$ (see the previous example). Consider a vertex $\vec{Z}_1 = z_1, z_2, \dots, z_n; +1$. The test verifies that \vec{Z}_1 is correctly located on the plot, i.e., that $G(\vec{Z}_1)$ plots in the correct half plane. Therefore, it verifies that $\vec{Z}_1^1 = -\vec{Z}_1^1 = -z_1, -z_2, \dots, -z_n; -1$ will plot in the opposite half plane. But since the test is complete, it verifies correct operation at all \vec{Z}_1 (i.e., all possible combinations of the z_1 through z_n), and therefore at all \vec{Z}_1^1 . Hence the test verifies correct operation of $G(\vec{Z})$.

Furthermore, it follows that if the test sequence for F consists of vertices $\vec{Z}_1, \vec{Z}_2, \dots, \vec{Z}_n$, then an equally good test sequence is $-\vec{Z}_1, -\vec{Z}_2, \dots, -\vec{Z}_n$. Of course, the latter cannot be actually applied, since z_0 is fixed (at +1) in any test.

To test the function H , related to F as prescribed above, we wish to verify the same $G(\vec{Z})$. However, the test cannot contain

certain values of \vec{Z} , namely those for which the i -th element in \vec{a}^F , a_i^F , corresponds to the modified threshold of H , a_0^H . Here the test must make $z_i = +1$. But if some vertex \vec{Z}_p in the test set of F has $z_i = -1$, it can be replaced by vertex $\vec{Z}_p^1 = -\vec{Z}_p$ without loss of information, and then \vec{Z}_p^1 will indeed have $z_i = +1$. It follows that given a test sequence for F , one can find the test sequence for H by making the appropriate choice between \vec{Z}_p and $-\vec{Z}_p$. This is the result we were seeking. The procedure for finding the test sequence for H given that for F is stated more precisely in the following algorithm, which is illustrated below.

ALGORITHM

Let the characteristic vector of F be $\vec{b}^F = b_1^F, b_2^F, \dots, b_n^F; b_0^F$ and that for H be $\vec{b}^H = b_1^H, b_2^H, \dots, b_n^H; b_0^H$, where $b_i^H = \pm b_j^F$.

1. List all the test vertices of F in $+1, -1$ notation, placing each entry under a column headed by the corresponding component of \vec{b}^F .
2. Under the component associated with the modified threshold place a $+1$ to augment each vertex. Add a column describing the output at each vertex.
3. Augment the list by adding to it each vertex and output multiplied by -1 .
4. Locate b_i^F such that $|b_i^F| = |b_0^H|$. If $b_i^F = b_0^H$, pick those vertices which are $+1$ in that column; otherwise pick those which are -1 .

5. Match the remaining b_i^H and b_j^F , multiplying columns by -1 if $b_i^H = -b_j^F$.

6. The vertices selected in step 4 and modified in step 5 are the test vertices of H.

Example: $F = ABC + ABD + ABE + ACD + ACE + BCD$, which has the characteristic vector $\vec{b}^F = 16, 12, 12, 8, 4; -8$. The minimal checking sequence for F is: 11001, 10101, 01110, 01101, 11000, 10011, and 10100. We wish to find the minimal test sequence for function $H = ab + ac + ad + ae + bc + bde + cd$, for which $\vec{b}^H = 16, 8, 12, 8, 4; 12$.

\vec{b}	16	12	12	8	4	-8	Step 5	Output	Test Vertex of H
F	A	B	C	D	E	b_0^F			
H	a	c	b_0^H	b	e	$-b_d^H$	d		
	1	1	-1	-1	1	1	-1	1	
	1	-1	1	-1	1	1	-1	1	←
	-1	1	1	1	-1	1	-1	1	←
	-1	1	1	-1	1	1	-1	-1	←
	1	-1	-1	1	1	1	-1	-1	
	1	1	-1	-1	-1	1	-1	-1	
	1	-1	1	-1	-1	1	-1	-1	←

	-1	-1	1	1	-1	-1	1	-1	←
	-1	1	-1	1	-1	-1	1	-1	
	1	-1	-1	-1	1	-1	1	-1	
	1	-1	-1	1	-1	-1	1	1	
	-1	1	1	-1	-1	-1	1	1	←
	-1	-1	1	1	1	-1	1	1	←
	-1	1	-1	1	1	-1	1	1	

The true test vertices of H are 10001, 01100, 00110, and 01011. The false test vertices are 00101, 10000, and 01010. Function F has 3 true and 4 false test vertices, while H has 4 true and 3 false ones.

To recapitulate, we have shown how the test sequence for H can be found from that for F by means of a very simple procedure. Consequently, to compile the optimum test sequence for all functions that are single-threshold-element realizable, one need find only one test for each essentially different characteristic vector. This is not very tedious for a reasonably small number of variables, since the number of essentially different characteristic vectors is not very large. (For six or less variables, Dertouzos has shown that there are 135 different characteristic vectors.)

REFERENCE

1. Dertouzos, M. L., "Threshold Logic: A Synthesis Approach", Cambridge, Mass.: MIT Press, 1965.

APPENDIX A

UPPER BOUND ON THE NUMBER OF PRIME IMPLICANTS OF UNATE FUNCTIONS

We wish to display all possible products of n variables in a matrix having the following form:

1. Each entry is distinct from every other entry and consists of a single product.
2. All products of length ℓ are placed in row ℓ .
3. If in a column there is a product of length ℓ and there is one or more products of length greater than ℓ , then there must be a product of length $\ell+1$ which differs from the product of length ℓ in precisely one variable.
4. All R entries in each row are placed in columns 1 through R .

We shall say that the matrix is "left-adjusted".

Matrices of the above form will be designated by $M(n)$. An example of a $M(3)$ matrix is given below:

$$\begin{bmatrix} A & B & C \\ AB & BC & AC \\ ABC \end{bmatrix}$$

A matrix $M(n)$ can be constructed from a matrix $M(n-1)$ by the following procedure:

1. In $M(n-1)$ create row n .
2. In each column C_j of $M(n-1)$, append in row $i+1$ product nP_i , where n is the n -th variable and P_i is the longest product

- (located in row i) in C_j . Call this modified matrix $M_1(n-1)$.
3. Append to the right of $M_1(n-1)$ a copy of $M(n-1)$ with the following changes:
- In each column delete P_i , defined above.
 - Multiply each of the other entries by n .
 - Place the modified entries in row R of $M(n-1)$ into row $R+1$ of the new matrix.
 - In column 1 and row 1, place the product consisting of the single variable n . Call the appended matrix $M_2(n-1)$.
4. Intersperse the columns of $M_2(n-1)$ with those of $M_1(n-1)$ such that to the right of a column there are no columns with a larger number of entries. At the end of this step $M(n)$ is completed.

Example: We construct $M(4)$ on the basis of $M(3)$ given in the previous example

$$\begin{array}{c}
 \text{Steps 1 and 2} \\
 \left[\begin{array}{ccc}
 1 & 2 & 3 \\
 A & B & C \\
 AB & BC & AC \\
 ABC & \textcircled{BCD} & \textcircled{ACD} \\
 \textcircled{ABCD} & \downarrow nP_2 & \downarrow nP_3 \\
 nP_1 & &
 \end{array} \right] = M_1(3)
 \end{array}$$

Step 3

1	2	3		1'	2'	3'
A	B	C		D		
AB	BC	AC		AD	BD	CD
ABC	BCD	ACD		ABD		
ABCD						
$M_1(3)$				$M_2(3)$		
$M(4)$						

Step 4. Not necessary, so that above matrix is $M(4)$. Note that $M(4)$ is left-adjusted.

We must now prove that the procedure results in a matrix which has all four of the properties previously listed. That the construction results in all $2^n - 1$ products possible (the product of zero length is not displayed) is clear from the construction, for $M_1(n-1)$ contains all the possible products not containing n and $M_2(n-1)$ together with the added entries nP_i in $M_1(n-1)$ lists all the possible products containing n . Similarly, it is easy to see that Steps 1 through 3 assure that properties 1 through 3 are preserved in $M(n)$, given that they exist in $M(n-1)$. What is not easy to see is that property 4 (left-adjustedness) is preserved by virtue of Step 4. To prove this, assume that $M(n-1)$ is left adjusted. We now show that $M(n)$ must also be left-adjusted. First note that because $M(n-1)$ is left-adjusted, a column of $M(n-1)$ with E entries will have the first (shortest) entry (of a continuous string) in row $\frac{n-E}{2}$. This is true for all columns but the first, because if there are E entries,

then there are $n-E$ products missing, one of which is of length zero and has no row in $M(n-1)$. Also, from the symmetry of the binomial coefficients and the left-adjustedness of $M(n-1)$, it follows that half of the missing entries are above the first existing entry and the other half are below the last entry. Now consider a column C with F entries in $M_2(n-1)$. With the exception of the first column, which in Step 3(d) has the product n added in row 1, column C arises from a column K of $F+1$ entries in $M(n-1)$ by virtue of Step 3(a). The first entry in column K occurs in row $\frac{n-(F+1)}{2}$ and, in accordance with Step 3(c), after multiplication by n , it will be placed in row $\frac{n-(F+1)}{2} + 1 = \frac{n-(F-1)}{2}$ of $M_2(n-1)$. But that is precisely the row of $M(n-1)$ in which columns with $F-1$ entries start. Because of Step 2, this is also the starting row for entries of length F in $M_1(n-1)$. Consequently, in both $M_2(n-1)$ and $M_1(n-1)$ columns with F entries start in the same row. Therefore, Step 4 will assure that $M(n)$ is left adjusted, and so property 4 is assured by the construction.

Lemma A.1: Matrix $M(n)$ has $\binom{n}{\lfloor \frac{n+1}{2} \rfloor} = N(n)$ columns.

Proof: Since $N(n)$ is the largest number of products of fixed length that can be formed from n variables and $M(n)$ is left adjusted, it follows that $M(n)$ has $N(n)$ columns.

Lemma A.2: No unate function of n variables can have more than $N(n)$ prime implicants.

Proof: Since $M(n)$ has property 3, each product in a column with more than one entry is either contained in or contains each of

the other products. Consequently, a unate function of n variables can have at most one prime implicant from each column, of which there are $N(n)$.

Lemma A.3: The symmetric function S of n variables which is true whenever $\left\lfloor \frac{n+1}{2} \right\rfloor$ or more out of n variables are true has the largest number of prime implicants.

Proof: By its definition, S has $N(n)$ prime implicants. By Lemma 2, no unate function can have more than $N(n)$ prime implicants.

Theorem A.1: The function S of Lemma A.3 and its complement, \bar{S} , together have the largest possible number of prime implicants.

Proof: Function \bar{S} is true whenever $n - \left\lfloor \frac{n+1}{2} \right\rfloor + 1$ or more of the n variables are false. (Note that the MSP expression for \bar{S} is unate in the complemented variables. For example, for a three variable function $S = AB + AC + BC$, and $\bar{S} = \bar{A}\bar{B} + \bar{A}\bar{C} + \bar{B}\bar{C}$.)

(a) n odd. Here $n - \left\lfloor \frac{n+1}{2} \right\rfloor + 1 = n - \frac{n+1}{2} + 1 = \frac{n+1}{2}$. Then S and \bar{S} have the same number of prime implicants, i.e., $N(n)$, which by Lemma A.2 cannot be exceeded by any other function. Hence the theorem holds.

(b) n even. Here $n - \left\lfloor \frac{n+1}{2} \right\rfloor + 1 = n - \frac{n}{2} + 1 = \frac{n}{2} + 1$. Now S has still the largest possible number of prime implicants, but \bar{S} has $N'(n) = \binom{n}{\frac{n}{2} + 1}$ prime implicants, and $N'(n) < N(n)$. Note that in $M(n)$, written in complemented variables only, the $N'(n)$ prime implicants of \bar{S} are precisely all the entries in row $\frac{n}{2} + 1$ of the matrix, which is the second longest row. Hence a function \bar{F}

with more prime implicants than \bar{S} would have to have products in row $\frac{n}{2}$, which are products of length $\frac{n}{2}$. For each product P in row $\frac{n}{2}$ that \bar{F} may have, the product formed by the remaining uncomplemented $\frac{n}{2}$ variables not used in P cannot be a prime implicant of F . (For example, let $n = 4$. Then if $\bar{A}\bar{B}$ is a prime implicant of \bar{F} , then CD cannot be a prime implicant of F .) It follows that by whatever number the prime implicants of \bar{F} exceed those of \bar{S} , the number of prime implicants of S exceed those of F . Hence the total number of prime implicants cannot exceed $N(n) + N'(n)$, and the theorem holds.

CHAPTER 3

AN ALTERNATIVE APPROACH TO CHECKING BASED ON A RESPONSE-TABLE

We consider the following two problems: A box is intended to mechanize a specified combinational function G , but if any malfunctions occur, the box will mechanize one of the specified combinational functions H , or I , ..., or W . By means of applying a set of input combinations, called a test, and making observations at the output terminals only, we wish to determine either

(a) that the box is working correctly, i.e., that it mechanizes G ,
or

(b) which one of the functions G, H, I, \dots, W it mechanizes.

If the intent is to answer (a), we say that the box is to be checked; here we expect a yes-no answer. If the intent is to answer (b), we say that the box is to be diagnosed; here we not only learn whether or not the box functions correctly, but also which malfunction class is present if the response is not according to G . (We assume that each malfunction class gives rise to one of the functions H, I, \dots, W .)

Clearly, diagnosis includes checking, and consequently a checking test need never be longer than a diagnosing test.

We shall assume that specifications G, H, I, \dots, W are such that no two are alike, i.e., for every pair of functions there exists at least one input combination to which they respond differently. Consequently, one can always perform diagnosis (and hence checking) by applying every possible input combination. However, it is desirable to make the set of inputs to be applied as small as possible. In the remainder of this

chapter we will develop techniques for finding "short" tests. Some of these will lead to the shortest possible test; others will not. Our results are very much like those given by W. H. Kautz (IEEE Trans. on Computers, EC-17, 4, April 1968, 352-366), but the point of view used is different.

3.1 ILLUSTRATIONS OF APPROACH

For each input combination, i , we define a partition P_i on the set of possible functions G, H, \dots, W according to the output. Two functions I and J are in the same block iff their response to i is identical.

Formally stated, for checking we wish to find a set S_c of input combinations such that the intersection of the associated partitions has circuit G in a block by itself and the remainder of the partition is arbitrary, i.e.,

$$\prod_{j \in S_c} P_j = (G, \text{arbitrary})$$

For diagnosing, we wish to find a set S_d of vertices such that

$$\prod_{j \in S_d} P_j = (0)$$

where (0) denotes the partition with a single element in each block.

Of course, the smallest checking (diagnosing) test is that for which the cardinality of S_c (S_d) is smallest.

Consider the input-output behavior described in Figure 3.1, where the row labels are the decimal equivalents of the input combinations (vertices), column G lists the response of the fault-free (good) circuit, and columns H through Q the response of each of the circuit variants which could arise due to a malfunction.* Note that the circuit has two

* Finding the functions H through W is not, in general, a short task, although it can always be done, if all malfunctions being considered result in combinational circuits.

binary outputs, so that the possible outputs range from 0 (00) to 3 (11). The figure also lists the partitions P_0 through P_7 , where blocks are marked off by commas rather than the conventional overscore.

	G	H	I	J	K	L	M	N	O	P	Q
0	3	3	3	2	2	2	3	3	3	3	3
1	2	2	0	1	1	2	3	3	2	2	1
2	0	0	0	0	0	0	0	0	0	0	0
3	2	1	1	2	2	1	2	2	2	0	0
4	0	0	0	0	1	1	1	0	0	0	0
5	0	0	0	1	1	1	0	0	0	0	1
6	1	1	2	3	3	3	2	1	0	1	1
7	2	2	0	1	1	2	3	3	2	2	1

- X $P_0 = (\text{GHIMNOPQ}, \text{JKL})$
- $P_1 = (\text{GHLOP}, \text{I}, \text{JKQ}, \text{MN})$
- X $P_2 = (\text{GHIJKLMNOPQ})$
- $P_3 = (\text{GJKMNO}, \text{HIL}, \text{PQ})$
- $P_4 = (\text{GHLJNOPQ}, \text{KLM})$
- $P_5 = (\text{GHIMNOP}, \text{JKLQ})$
- $P_6 = (\text{GHNPO}, \text{IM}, \text{JKL}, \text{O})$
- X $P_7 = (\text{GHLOP}, \text{I}, \text{JKQ}, \text{MN})$

Fig. 3.1 Example 1

In scanning these partitions, we observe that P_2 places all the states in one block. Hence applying vertex 2 gives no information as to which of the circuits G through Q is actually at hand. Consequently, no test need include vertex 2, and we eliminate P_2 from further consideration.

We also observe that P_1 and P_7 are identical, and consequently no test set need contain both vertex 1 and vertex 7. We therefore eliminate P_7 from further consideration.

Next, we observe that by combining the first, second, and fourth block of P_6 , we obtain P_0 . This means that whatever information is obtained from applying vertex 0 is surely obtained from applying vertex 6. We say that vertex 6 "covers" vertex 0, and eliminate P_0 from further consideration.

The partitions eliminated so far have been marked by X in Figure 3.1.

We now observe that P_3 is the only partition which can differentiate between G and H. Hence any test must contain vertex 3, since both a checking test and a diagnosing test must be able to resolve between circuits G and H, because G is the only circuit which functions correctly. We say that P_3 is "essential," because any test set must contain vertex 3.

Next, we observe that P_1 is also essential, because it is the only remaining partition which differentiates between G and N. (Recall that P_7 has been eliminated from consideration.) Also, P_6 is the only partition that can differentiate between G and O, and is therefore essential.

Since any test must include vertices 1, 3, and 6, it is useful to form the intersection

$$\begin{aligned} P_1 \cdot P_3 \cdot P_6 &= (G,H,I,JK,L,M,N,O,P,Q) \\ &= P'_6 \end{aligned}$$

If our objective is to check only, then, because P'_6 is of the form (G, arbitrary), the shortest test is $S_c = \{1,3,6\}$; on the other hand, for diagnosing we obtain as the shortest test $S_d = \{1,3,4,6\}$, because

$P_4 \cdot P'_6 = (0)$. While every diagnosing test is a checking test, it is not true that the shortest checking test is contained in the shortest diagnosing test, as a counterexample given in Section 3.3 will show.

As a second example, consider a three-input, single-output circuit which has the partitions given in list L_0 of Figure 3.2. (This example is given by Kautz.)

L_0	$\begin{matrix} P_0 \\ L_1 \end{matrix}$	$\begin{matrix} P_2 \\ L_1 \end{matrix}$
$P_0 = (GIL, HJKMN)$		$P''_0 = P_0 \cdot P_2 = (GI, L, KM, HJN)$
$P_1 = (GHM, IJKLN)$	$P'_1 = P_1 \cdot P_0 = (G, IL, HM, JKN)$	$P''_1 = P_1 \cdot P_2 = (GM, IK, H, JLN)$
$P_2 = (GIKM, HJLN)$	$P'_2 = P_2 \cdot P_0 = (GI, KM, L, HJN)$	
$P_3 = (GHIK, JLMN)$	$P'_3 = P_3 \cdot P_0 = (GI, HK, L, JMN)$	$P''_3 = P_3 \cdot P_2 = (GIK, H, M, JLN)$
$P_4 = (GIJKLMN, H)$		
$P_5 = (GHK, IJLMN)$	$P'_5 = P_5 \cdot P_0 = (G, HK, IL, JMN)$	$P''_5 = P_5 \cdot P_2 = (GK, IM, H, JLN)$
$P_6 = (GHIJ, KLMN)$	$P'_6 = P_6 \cdot P_0 = (GI, HJ, L, KMN)$	$P''_6 = P_6 \cdot P_2 = (GI, HJ, KM, LK)$
$P_7 = (GHIMN, JKL)$	$P'_7 = P_7 \cdot P_0 = (GI, JK, L, HMN)$	$P''_7 = P_7 \cdot P_2 = (GIM, K, HN, JL)$

Figure 3.2 Example 2

We can make the following observations:

- (1) No vertex places all circuits in one block.
- (2) No pair of partitions is identical.
- (3) No partition is covered by any other partition.
- (4) There are no essential partitions.

It turns out, however, that P_4 can be eliminated on the basis of the following argument. Partition P_4 distinguishes only between circuit H and any one of the others (G, I, J, \dots, N) , but is not essential. Consequently, if vertex 4 were used in a test, then it could be replaced by some other single vertex without lengthening the sequence.

At this point, we have no further steps for eliminating partitions and are now faced with a classical covering problem: we must find that subset of partitions which is smallest and can simultaneously differentiate between all possible circuit label pairs (for the case of diagnosis), or between all possible pairs in which G is one of the elements (for the case of checking).

To ease the task on hand, we observe that only to differentiate between labels G and H, we must apply either vertex 0 or 2. The partitions resulting from either choice are also shown in Figure 3.2 with list $L_1^{P_0}$ indicating the result of choosing P_0 and $L_1^{P_2}$ the result of choosing P_2 . In neither list is there an identical partition pair, nor is there a partition covered by another partition, since all have the same number of blocks. From $L_1^{P_0}$ we observe that there are two checking tests of length two: vertices 0 and 1 form one checking test, and vertices 0 and 5 a second.

Continuing with Example 2, we now find a diagnosing test. Scanning lists $L_1^{P_0}$ and $L_2^{P_2}$, we observe that in all partitions except P_6'' there is a block with three elements. But P_0 through P_7 are all two-block partitions. Consequently, if there exists a diagnosing test of length three, then that test must contain vertices 6 and 2. But then there must exist a single partition which can simultaneously differentiate between pairs GI, HJ, KM, and LN. Now only P_1 and P_5 can differentiate between G and I. We form

$$P_6''P_1 = (G,H,I,J,K,LN,M) = P_6'''$$

$$P_6''P_5 = (G,H,I,J,K,LN,M) = P_6'''$$

Hence, a minimum diagnosing test is specified by P_6'' together with any vertex that can distinguish between L and N, viz., 0 or 7. Consequently, the following are some of the minimum-length diagnosing tests:

(a) 6,2,1,0

(b) 6,2,1,7

(c) 6,2,5,0

(d) 6,2,5,7

This example is quite typical. For circuits that are not very complex, i.e., have a moderate number of inputs and not very many variants due to malfunctions, a short, or even shortest, checking test can be found quite readily. Short diagnosing tests, on the other hand, are more difficult to find, particularly if minimum test length is desired. In fact, for the case of checking tests we can simplify the procedure followed in the previous example, as described in the next section.

3.2 FINDING SHORTEST CHECKING TESTS

For the purpose of finding a checking test, we can define a broader notion of coverage, called C-coverage: a partition P_i C-covers another partition P_j if the block of P_i containing G is contained in a block of P_j .

Returning to the example of Figure 3.2 we observe that P_0 C-covers P_4 , P_1 C-covers P_7 , and P_5 C-covers P_3 . This leaves for consideration the five partitions listed below

$$P_0 = (GIL, HJKMN)$$

$$P_1 = (GHM, IJKLN)$$

$$P_2 = (GIKM, HJLN)$$

$$P_5 = (\text{GHK}, \text{IJLMN})$$

$$P_6 = (\text{GHIJ}, \text{KLMN})$$

Observe again that only P_0 and P_2 can differentiate between G and H. Because in both P_0 and P_2 states J and N are in the same block with state H, it follows that any checking sequence that differentiates between G and H also differentiates between G and J, and between G and N. To simplify our future work, we can therefore omit J and N in all partitions and obtain

$$P'_0 = (\text{GIL}, \text{HKM})$$

$$P'_1 = (\text{GHM}, \text{IKL})$$

$$P'_2 = (\text{GIKM}, \text{HL})$$

$$P'_5 = (\text{GHK}, \text{ILM})$$

$$P'_6 = (\text{GHI}, \text{KLM})$$

Now to differentiate between G and I, either P'_1 or P'_5 must be used. In either case, G will also be differentiated from L, and so we omit L from further consideration and obtain

$$P''_0 = (\text{GI}, \text{HKM})$$

$$P''_1 = (\text{GHM}, \text{IK})$$

$$P''_2 = (\text{GIKM}, \text{H})$$

$$P''_5 = (\text{GHK}, \text{IM})$$

$$P''_6 = (\text{GHI}, \text{KM})$$

Now P''_0 C-covers P''_2 and P''_6 . Eliminating the latter two partitions makes P''_0 C-essential. We therefore form

$$P'''_1 = P''_1 P''_0 = (\text{G}, \text{I}, \text{HM}, \text{K})$$

$$P'''_5 = P''_5 P''_0 = (\text{G}, \text{I}, \text{HK}, \text{M})$$

Hence there exist at least two minimum-length checking tests: $\{0,1\}$ and $\{0,5\}$. This result was found before, but with less pain in the second development.

We summarize our approach to the determination of minimum-length checking tests in the following algorithm, where n denotes the number of binary inputs, and m the number of binary outputs. The execution of the algorithm is stopped when a partition of the form $(G, \text{arbitrary})$ is obtained.

1. For the correctly functioning circuit G and each of its variants H, I, \dots, Z due to a malfunction, record in a column the response due to each possible input combination (input vertex) j , $0 \leq j \leq 2^n - 1$. The result is called a response table, an example of which was given in Fig. 3.1.

2. From the response table, form the set of all partitions P_j . Two circuits labelled Q and R are placed in the same block of P_j iff their response to input vertex j is identical. Partition P_j will have at most 2^m blocks.

3. Eliminate all partitions which have a single block, i.e., all $P_j = (I)$.

4. If $P_i = P_j = \dots = P_q$, eliminate all but one of these, say P_i .

5. Eliminate the C -covered partitions. (A partition P_i is C -covered by another partition P_j if the elements in the block of P_j containing G are a subset of the elements in the block of P_i containing G .)

6. Find the C -essential partitions. (A partition is C -essential if it is the only one which has circuits G and X , any other circuit, not in the same block.)

7. Eliminate all two-block partitions that are not essential and have a singleton K , $K \neq G$, as one block.

8. Intersect each of the surviving partitions with the intersection of all essential partitions. Call all the resulting partitions the current list of partitions.

9. Repeat Steps 4-8 on the current list of partitions in any order and as frequently as possible until they can no longer be applied.

10. Find a small subset of partitions P_a, P_b, \dots, P_c which are the only ones that distinguish between circuit G and some one other circuit, X . Determine all those circuit labels X, Y, \dots, Z which in all of these partitions are not in the block containing label G . Eliminate labels Y through Z in all partitions.

11. Repeat Steps 3 through 10 as often as possible.

12. Arbitrarily select one partition P_s for inclusion in the test. Intersect all remaining partitions with P_s .

13. Repeat Steps 3 through 11.

14. Repeat Steps 12 and 13 until a checking test is found and the number of partitions used is a minimum.

The major computational labor arises from Steps 12 through 14, for here we go through a classical decision tree and our objective is to find that set of branches which leads to a minimum solution. Since this type of computation is well known, we need not dwell on it.

3.3 FINDING DIAGNOSING TESTS

To find a minimum-length diagnosing test, the procedure is similar to that given above for finding minimum checking tests. The differences are these:

(a) In Step 5, replace C-covered by D-covered. A partition P_i is D-covered by another partition P_j if P_i is an enlargement of P_j .

(b) In Step 6, replace C-essential by D-essential. A partition is D-essential if it is the only one which differentiates between circuits X and Y, for any X and any Y.

(c) In Step 7, K is not restricted in any way.

(d) Step 10 and therefore Step 11 are not applicable.

We illustrate the process of finding an optimum diagnosing test starting with Step 2.

2. $P_0 = (\text{GHIKLMNOPQ}, \text{J})$

$P_1 = (\text{GIJMNPQ}, \text{HL}, \text{KO})$

$P_2 = (\text{GIJLNOQ}, \text{HM}, \text{KP})$

$P_3 = (\text{GIJLMOP}, \text{HN}, \text{KQ})$

$P_4 = (\text{GHKNQ}, \text{IOP}, \text{JLM})$

$P_5 = (\text{GHKMP}, \text{IOQ}, \text{JLN})$

$P_6 = (\text{GHJKLP}, \text{IPQ}, \text{MN})$

$P_7 = (\text{GHIJLMN}, \text{KOPQ})$

3. None

4. None

5. None

6. None

7. Eliminate P_0

8. None

9. Returning to Step 6, we find that P_1 is now D-essential, because only P_1 differentiates between J and L.

$$10. P'_2 = P_2 \cdot P_1 = (GIJNQ, M, P, H, L, KO)$$

$$P'_3 = P_3 \cdot P_1 = (GIJMP, N, Q, H, L, K, O)$$

$$P'_4 = P_4 \cdot P_1 = (GNQ, IP, JM, H, L, K, O)$$

$$P'_5 = P_5 \cdot P_1 = (GMP, IQ, JN, H, L, K, O)$$

$$P'_6 = P_6 \cdot P_1 = (GJ, IPQ, MN, HL, KO)$$

$$P'_7 = P_7 \cdot P_1 = (GIJMN, PQ, HL, KO)$$

11. None

12. We observe that only P_4 and P_5 can differentiate between G and

J. Suppose P'_4 is selected for inclusion in the test.

$$P''_2 = P'_2 \cdot P'_4 = (GNQ, I, J, M, P, H, L, K, O)$$

$$P''_3 = P'_3 \cdot P'_4 = (G, IP, JM, N, Q, H, L, K, O)$$

$$P''_5 = P'_5 \cdot P'_4 = (G, M, P, I, Q, J, N, H, L, K, O) = (0)$$

$$P''_6 = P'_6 \cdot P'_4 = (G, J, IP, Q, M, N, H, L, K, O)$$

$$P''_7 = P'_7 \cdot P'_4 = (GN, Q, I, P, JM, H, L, K, O)$$

Thus a minimum-length diagnosing test is 1,4,5 .

If we had selected P'_5 for inclusion in the test, there would result

$$P''_2 = P'_2 \cdot P'_5 = (G, M, P, IQ, JN, H, L, K, O)$$

$$P''_3 = P'_3 \cdot P'_5 = (GMP, I, Q, J, N, H, L, K, O)$$

$$P''_4 = P'_4 \cdot P'_5 = (0)$$

$$P''_6 = P'_6 \cdot P'_5 = (G, J, IQ, P, M, N, H, L, K, O)$$

$$P''_7 = P'_7 \cdot P'_5 = (GM, P, I, Q, JN, H, L, K, O)$$

Since $P''_4 = (0)$, we again find the same minimum-length diagnosing test, which happens to be unique in this example.

This example also illustrates that a minimum-length checking test is not necessarily contained in any minimum-length diagnosing test.

It is readily verified that there are only two checking tests of length

two for this example: 3,4 and 2,5 . Obviously, neither is contained in the unique minimum-length diagnosing test.

Yet if we form

$$P'_1 = P_1 \cdot P_3 \cdot P_4 = (G,N,Q,IP, JM,H,L,K,O)$$

we observe that $P'_1 P_2$ is a diagnosing test (of length 4). Experience with a limited number of examples indicates that to avoid the considerable labor involved in finding optimum diagnosing sequences, it may well be good strategy to start with a minimum-length checking test (which is not so hard to find) and then to enter the algorithm for finding diagnosing tests with the partitions involved in the checking sequence as essential. This approach seems sensible for actual applications, for one will often conduct a test without knowing whether or not the circuit is faulty, and one can expect that the circuit is more likely to be fault-free than not. In the majority of cases then, the initial checking test need not be followed by the additional inputs that complete the diagnosing test, and therefore the non-minimum length is not so crucial.

A major contribution to the increase in computational complexity for a diagnosing test comes about from the fact that in Step 6, the number of pairs of circuits that must be considered is large: for a total of z circuits, there are $z(z-1)/2$ pairs to be investigated.

3.4 CONCLUSIONS

We have given techniques for finding checking and diagnosing tests and found the methods for determining optimum checking tests more conveniently executable than those for determining optimum diagnosing tests. The difference comes about primarily because (a) in Step 6 checking involves consideration of only $z-1$ circuit pairs (versus

$z(z-1)/2$ for diagnosing) and (b) Step 10 simplifies the procedure for the case of checking.

Furthermore, the compilation of the response table (see Step 1) is, for a reasonably complex circuit, a time-consuming process.

Two alternatives appear to be open for easing the computational task:

1. Approximate methods for finding checking sequences, such as given in Chapter 1.

2. Designing circuits for ease of checking and diagnosability.

The latter appears to be practicable at this time, due to the advent of LSI, and will form the subject of a future investigation.

SECTION IV

MULTI-VALUED LOGIC

C. Ying and A. K. Susskind

This section is concerned with logic in which a physical signal lead can carry more than two signal levels. A technique is described for the synthesis of efficient combinational networks which is applicable no matter what the number of signal levels is, based on building blocks that can be readily realized. Finally, the theory of symmetric functions is extended to the case of multi-valued signals.

CHAPTER 1

INTRODUCTION

The topic of multivalued logic has been studied, as one might suspect, by both logicians and engineers. The logicians are mainly concerned with the formulation of calculi analogous to the conventional (binary) propositional and predicate calculi. Post¹ generalized binary operations to m -valued ones and showed functional completeness - the ability of the set to realize any of the m^m functions of n variables. Rosser and Turquette² concerned themselves with definition of truth and falsity in m -valued logics, preservation of modus ponens, and problems of consistency and completeness (in the Godel completeness sense) of their logics. More recently, diForino³ approached m -valued logic from the point of view of algebraic logic.

To the logic designer interested in m -valued switching, most of these investigations are rather impertinent. The first thorough in-depth discussion of m -valued switching was presented by Lowenschuss⁴ in 1958. From the logicians he borrowed the concept of functional completeness and recommended Slupecki's theorem⁵ as an easy test for completeness of a set of proposed operators.

Slupecki's theorem, being so extremely powerful, will be restated here without proof:

Theorem 1.1: A set of m -valued operators S is functionally complete iff

- 1) S can realize all m^m functions of one variable
- 2) S can realize one two-variable function $f(x,y)$ such that:

- a) the range of f is the whole alphabet $A = \{0, 1, \dots, m-1\}$
- b) there exist a, b, c, d in A , such that
 - i) $f(a, b) \neq f(a, c)$
 - ii) $f(a, b) \neq f(a, d)$
 - iii) $f(a, c) \neq f(a, d)$

For logical designers, however, a far more constructive proof of the completeness of a set of m -valued operators is simply an algorithm which describes an effective way to generate realizations for any of the m^m functions in m -valued switching. Such an algorithm is usually expressed in the form of an expansion theorem:

$$f(x_1, \dots, x_n) = F[x_1, f(c, x_2, \dots, x_n)]$$

where c is any truth value, and F some functional. For example, the well-known expansion theorem for the and-or-not set is

$$f(x_1, \dots, x_n) = x_1 f(1, x_2, \dots, x_n) + \bar{x}_1 f(0, x_2, \dots, x_n)$$

which leads to the canonic sum-of-products form and immediate realization for any of the 2^{2^n} functions.

Recently, there has been an increase in interest among logic designers in multi-valued switching. However, by far the majority limited themselves to ternary switching. We are not aware of any systematic approach to the entire problem of multivalued switching - both the theories and their implementations. The study by Yoeli and Rosenfeld⁶ limited itself to the ternary case and considered only one particular system of logic. Lee and Chen⁷, Berlin⁸, and Gazale⁹ considered algebraic aspects only, without any concern as to circuit realization. Lowenschuss and Muhldorf¹⁰ both gave excellent discussions

of the problem, showing their awareness of and interest in the applications of m-valued logics.

In our study of multivalued switching systems, we have found that there are essentially two approaches to the problem of finding m-valued switching systems. First is the algebraic approach, exemplified by the systems due to Post, Rosser and Turquette, and Muhldorf. The second approach to multivalued switching is from a device point of view. Wigington's parametric oscillator¹¹ system represents such an approach. The problem with the device approach is obvious - it almost always leads to ad hoc synthesis procedures, if at all. That is to say, procedures for which minimality, uniqueness, and convergence are not guaranteed. The complexity of such a system is often the only property that is guaranteed. In our previous report, we indicated how threshold logic can be extended to the m-valued case and indicated an effective synthesis procedure leading to a simple two-level form of realization. Further work along these lines is reported in Ref. 18.

Any logic designer knows the desirability of algebraic structure in a logic system. The Boolean nature of and-or-not makes the system easy to manipulate. The algebraic approach to multivalued switching, then, is to construct systems with such desirable algebraic structure. The Muhldorf system discussed in Chapter 2 is an excellent illustration of the advantages of such an approach for the multilevel case. Of all the systems investigated, the one based on Muhldorf's work is easiest to handle and while procedures leading to optimum circuit forms were not found, nevertheless the algorithm given in Chapter 2 leads to "good" realizations.

In Chapter 3, we extend the theory of symmetric functions to the m -valued case and give an effective procedure for detecting symmetry. It is noteworthy that as the number of levels increases to five or more, one must differentiate between two types of symmetry, only one of which has the desired property of leading to circuit realizations that are independent of the lead assignment.

CHAPTER 2

THE MUHLDORF SYSTEM OF MULTI-LEVEL SWITCHING

Muhldorf proposed a system of m-level switching which is a generalization of the Boolean operators in two-valued logic. Muhldorf¹³ himself applied his system to ternary switching. The following is a concise development of his system for m-level switching and an effective procedure for using it in synthesis.

We shall find it convenient to call the set of constants $0, 1, \dots, m-1$ the set A . By A^n will be meant the n^{th} Cartesian product of A .

Muhldorf defined the following $2m$ allowable operations:

- (1) The maximum operator (+)

$$x + y \triangleq \max(x, y)$$

- (2) The minimum operator (.)

$$x \cdot y \triangleq \min(x, y)$$

Although the operators defined are binary, the associativity of the operations allows us to write unambiguously:

$$\sum_{i=1}^n x_i \triangleq x_1 + x_2 + \dots + x_n = \max(x_1, \dots, x_n)$$

$$\prod_{i=1}^n x_i \triangleq x_1 \cdot x_2 \cdot \dots \cdot x_n = \min(x_1, \dots, x_n)$$

Note that the operators $\cdot, +$ induce a distributive lattice structure on A . Distributivity follows from the properties of the max- and min-operators:

$$x + yz = (x + y)(x + z)$$

$$(x + y)z = xz + yz$$

(3) The best way to describe the remaining $2m-2$ operators, which are unary, is by displaying their truth-tables:

x	x^0	x^{01}	$x^{01\dots i}$	$x^{01\dots m-2}$
0	$m-1$	$m-1$	$m-1$	$m-1$
1	0	$m-1$	$m-1$	$m-1$
.	.	0	.	.
.
.
i	0	0	$m-1$	$m-1$
.	.	.	0	.
.
.
$m-2$	0	0	0	$m-1$
$m-1$	0	0	0	0

$m-1$ "step-down" functions

x	x^{m-1}	$x^{(m-2)(m-1)}$	$x^{i(i+1)\dots(m-1)}$	$x^{12\dots(m-1)}$
0	0	0	0	0
1	0	0	0	$m-1$
.
.
.	.	.	0	.
i	0	0	$m-1$	$m-1$
.
.
.	.	0	.	.
$m-2$	0	$m-1$	$m-1$	$m-1$
$m-1$	$m-1$	$m-1$	$m-1$	$m-1$

$m-1$ "step-up" functions

Our interest in the Muhldorf system arises from the case with which the $2m$ operations can be implemented with semiconductor devices. The

max and min functions can be realized by the diode networks shown in Figs. 2.1 a and b, respectively. Levels V_+ and V_- signify constant signals higher than V_{m-1} and lower than V_0 , respectively. Note that for binary signals, these networks correspond to Boolean OR and AND gates, when positive logic is assumed. The unary operators are threshold detectors and can be realized in the form of Fig. 2.2. The signal at A corresponds to one of the step-down operators and that at B to one of the step-up operators. Resistors R_1 and R_2 and source V determine the level of the threshold, i.e., the lowest potential at X which causes the left transistor to conduct. When the left transistor conducts (is cut off) the right transistor is cut off (conducts). Furthermore, when a transistor conducts, we assume that its collector potential is essentially V_0 , whereas if it is cut off, the collector potential is essentially V_{m-1} . If values are so chosen that point A realizes $x^{01\dots i}$ because the left transistor is cut off for all input potentials V_i or less, then point B will realize $x^{(i+1)(i+2)\dots(m-1)}$.

Certain unary functions play an important role in the demonstration of the functional completeness of the set of $2m$ operations defined above. We shall single them out as follows:

$$x^i \triangleq \begin{cases} m-1 & \text{if } x = i \\ 0 & \text{if } x \neq i \end{cases} \quad i = 0, 1, \dots, m-1$$

We can realize each x^i as follows:

$$x^i = x^{01\dots i} \cdot x^{i(i+1)\dots(m-1)}$$

Reference to the truth tables will show the validity of the equation.

Now we are ready to prove the following

Theorem 2.1: The $2m$ operations defined above, together with the constants A, can generate any m -valued function of n -variables.

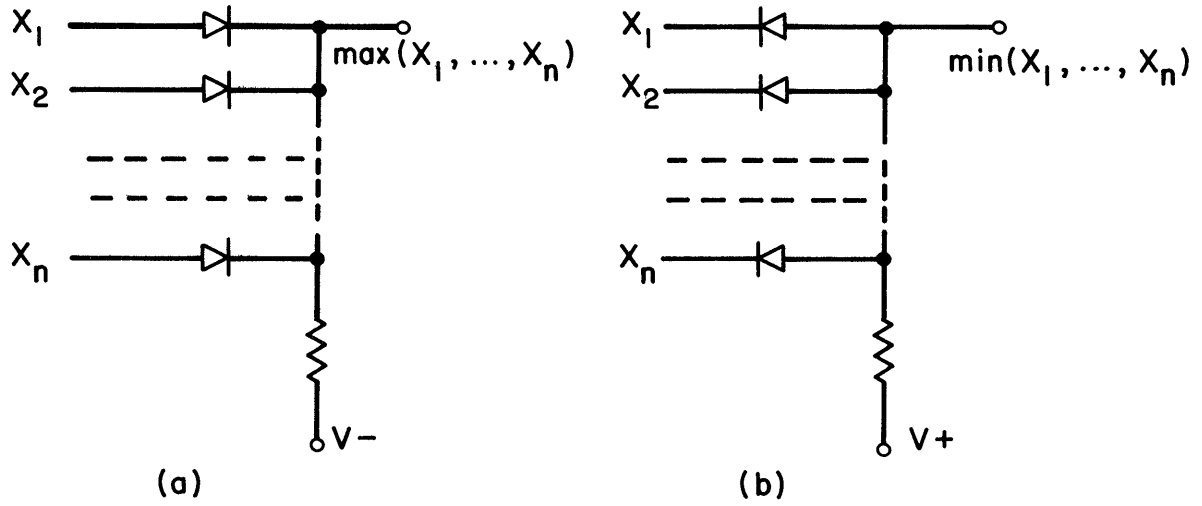


Fig. 2.1 Networks for Max and Min Gates

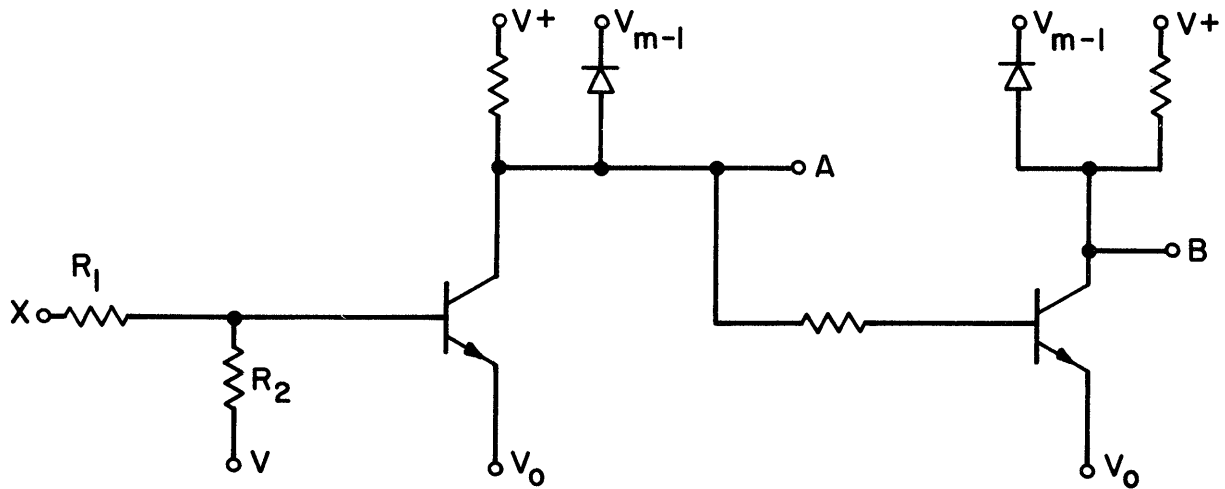


Fig. 2.2 Muhldorf Unary Gates

Proof: It suffices to show that for any m-valued function $f(x_1, \dots, x_n)$ the following claim is valid.

$$\text{Claim: } f(x_1, \dots, x_n) = \sum_{(a_1, \dots, a_n) \in A^n} f(a_1, \dots, a_n) \cdot x_1^{a_1} \cdot x_2^{a_2} \dots x_n^{a_n}$$

Proof of Claim: As the inputs x_1, \dots, x_n take on any of the m^n possible sets of values in A^n , say $(x_1, x_2, \dots, x_n) = (a_1, a_2, \dots, a_n)$, only one term in the above "sum" will be non-zero, namely the term

$$f(a_1, \dots, a_n) \cdot x_1^{a_1} \cdot x_2^{a_2} \dots x_n^{a_n}$$

which at $x_1 = a_1, \dots, x_n = a_n$ is just $f(a_1, \dots, a_n)$, as claimed.

The resemblance of the above to the two-valued, standard sum-of-products form explains the choice of notation for the max- and min-operators.

In fact, if one defines

$$\bar{x}^i \triangleq \begin{cases} 0 & \text{if } x = i \\ m-1 & \text{if } x \neq i \end{cases}$$

with the realization

$$\bar{x}^i = x^{01 \dots (i-1)} + x^{(i+1) \dots (m-1)},$$

then a standard product-of-sums form can also be found for any m-valued function f of n variables.

$$f(x_1, \dots, x_n) = \prod_{(a_1, \dots, a_n) \in A^n} (f(a_1, \dots, a_n) + x_1^{\bar{a}_1} + x_2^{\bar{a}_2} + \dots + x_n^{\bar{a}_n})$$

In this case, at each vertex in A^n only one of the "sums" in the "product" can have value less than $m-1$. Hence the whole expression takes on that value, which is exactly the value taken by f at that vertex.

Example:

	x		
y	0	1	2
0	2	2	0
1	0	0	1
2	0	1	0

Table for f

For the truth table shown, the standard sum-of-products realization would be:

$$f(x,y) = (2 \cdot x^0 \cdot y^0) + (2 \cdot x^1 \cdot y^0) + (1 \cdot x^2 \cdot y^1) + (1 \cdot x^1 \cdot y^2)$$

In the standard product-of-sums form,

$$f(x,y) = (2 + x^{\bar{0}} + y^{\bar{0}}) (2 + x^{\bar{1}} + y^{\bar{0}}) (1 + x^{\bar{2}} + y^{\bar{1}}) (1 + x^{\bar{1}} + y^{\bar{2}})$$

Getting back to the sum-of-products form, because of distributivity, we can represent any function of n m-valued variables in the form

$$f(X) = s_{m-1}(X) + (m-2) \cdot s_{m-2}(X) + \dots + 1 \cdot s_1(X)$$

where each function s_i has value m-1 for all those input combinations where f has value i and s_i has value 0 otherwise. Hence the previous example can be expressed in the above form as

$$f(x,y) = x^0 \cdot y^0 + x^1 \cdot y^0 + 1(x^2 \cdot y^1 + x^1 \cdot y^2)$$

We now observe that in the above representation it is not necessary to restrict the s_i as severely as was stated. For if s_i equals some arbitrary value for some input combination C for which $s_j = m-1$ with $j > i$, then $j s_j$ will contribute an output of j, whereas $i s_i$ contributes an output of i or less. Since $j > i$, $f(C) = j$ no matter what s_i is. This leads to the following form for any function:

$$f(X) = g_{m-1}(X) + (m-2) g_{m-2}(X) + \dots + 1 g_1(X)$$

where

$$g_i(X) = \begin{cases} 0 & \text{if } f(X) < i \\ m-1 & \text{if } f(X) = i \\ \emptyset & \text{if } f(X) > i \end{cases}$$

and \emptyset stands for "don't-care", i.e., may be arbitrarily specified. The judicious use of the \emptyset values forms the central part of the synthesis procedure that follows.

Yoeli and Rosenfeld⁶ studied the minimization problem for the case $m=3$ and showed that the minimization procedure of Scheinman (Ref. 15) is applicable. Here, we point out that the procedure is also applicable to the case of arbitrary m and will summarize it below. The underlying reason for the applicability of the Scheinman method is that we can make the g -functions bivalued: either $m-1$ or 0 . Because of the existence of the don't-care conditions, however, much choice is left and we will suggest one approach that appears to handle the don't-care conditions effectively, although not optimally.

It will be our objective to recast each of the g_i in the form of a sum of products with each sum having a small (but not necessarily minimum) number of literals. But whereas in the two-level form of bivalued functions one can have only three kinds of literals (x_i, \bar{x}_i , or no appearance of x_i), in the m -valued case there are $\frac{m(m+1)}{2}$ kinds of literals:

$x_i^0, x_i^1, \dots, x_i^{m-1}$ m ways
 $x_i^{01}, x_i^{12}, x_i^{23}, \dots, x_i^{(m-2)(m-1)}$... m-1 ways
 $x_i^{012}, x_i^{123}, \dots, x_i^{(m-3)(m-1)(m-1)}$... m-2 ways
 \vdots
 $x_i^{012\dots(m-2)}, x_i^{123\dots(m-1)}$ 2 ways
 $x_i^{012\dots(m-1)}$ = independent of x_i ... 1 way

The Scheinman procedure may be recast as follows:

1. Form a list L_0 of the vertices given. Put a check next to the don't-care entries.
2. Form $\frac{m(m+1)}{2}$ empty lists with the labels $x_1^0, x_1^1, \dots, x_1^{12\dots(m-1)}, x_1^{01\dots(m-1)}$.
3. Enter a $(n-1)$ -tuple $a_2 \dots a_n$ into the list with heading x_1^i iff $a_2 \dots a_n$ is a vertex in L_0 . If $a_2 \dots a_n$ is a don't-care vertex in L_0 (i.e., it is checked) also check the new entry in list x_1^i .
4. From the m lists x_1^0, \dots, x_1^{m-1} formed in Step 3 enter an entry $a_2 a_3 \dots a_n$ in list $x_1^{i(i+1)}$ iff that entry is found in lists x_1^i and x_1^{i+1} and place a check mark next to entry $a_2 a_3 \dots a_n$ in lists x_1^i and x_1^{i+1} .
5. Make entry $a_2 a_3 \dots a_n$ in list $x_1^{i(i+1)\dots(j-1)(j)}$ iff that entry is both in lists $x_1^{i(i+1)\dots(j-1)}$ and list $x_1^{(i+1)\dots(j-1)j}$. Again check entries copied into successor lists. Repeat until no further copying can be done.
6. Remove all empty lists and lists with all entries marked with a check. For each of the remaining lists of $(n-1)$ -tuples reiterate steps 2, 3, 4, and 5, using as column headings $x_2^0, x_2^1, \dots, x_2^{01\dots(m-1)}$.

7. Next, process all surviving lists according to step 6, first with x_3 headings, then with x_4 headings, and finally with x_{n-1} headings.
8. Finally, at the $(n-1)$ th iteration we are dealing with 1-tuple lists, which we can immediately write down as a term involving x_n . To these terms are "multiplied" all the list headings that they derived from. The sum of all these products yields the desired expression.

Example: We illustrate the above algorithm by the following simple example for $m=4$.

	$x_1 x_2 x_3 x_4$	
L_0 list:	0 0 1 2	1 1 2 2
	0 0 1 3 ✓	1 3 1 1
	0 0 2 2	1 3 1 2 ✓
	0 0 2 3 ✓	
	0 1 1 2	
	0 1 1 3	
	0 1 2 2	
	0 1 2 3	
	1 0 1 2	
	1 0 1 3	
	1 0 2 2	
	1 0 2 3 ✓	
	1 1 0 1	
	1 1 0 2	
	1 1 1 1	
	1 1 1 2	
	1 1 1 3 ✓	
	1 1 2 3 ✓	
	1 2 0 1	
	1 2 0 2	
	1 2 1 1	
	1 2 1 2	
	1 3 0 1	
	1 3 0 2	

The next step is to obtain the $\frac{4 \cdot 5}{2} = 10$ lists as follows:

x_1^0 list is

- 012 ✓
- 013 ✓
- 022 ✓
- 023 ✓
- 112 ✓
- 113 ✓
- 122 ✓
- 123 ✓

x_1^1 list is

- 012 ✓
- 013 ✓
- 022 ✓
- 023 ✓
- 101
- 102
- 111
- 112 ✓
- 113 ✓
- 122 ✓
- 123 ✓
- 201
- 202
- 211
- 212
- 301
- 302
- 311
- 312

x_1^{01} list is

- 012
- 013
- 022
- 023
- 112
- 113
- 122
- 123

Notice that the x_1^0 list, being all checked, is erased.

We now reiterate the expansion step on the list x_1^1 obtaining the following lists:

x_2^0 list

- 12 ✓
- 13 ✓
- 22 ✓
- 23 ✓

x_2^1 list

- 01 ✓
- 02 ✓
- 11 ✓
- 12 ✓
- 13 ✓
- 22 ✓
- 23 ✓

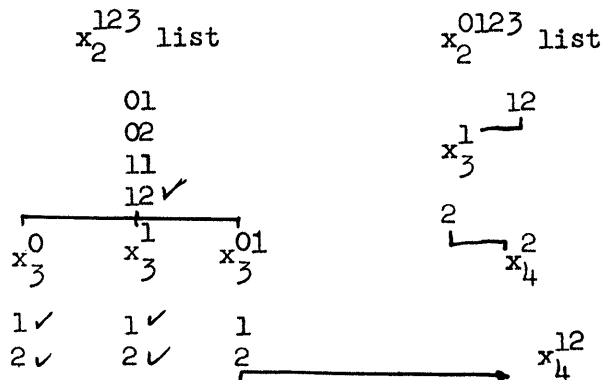
x_2^2 list

- 01 ✓
- 02 ✓
- 11 ✓
- 12 ✓

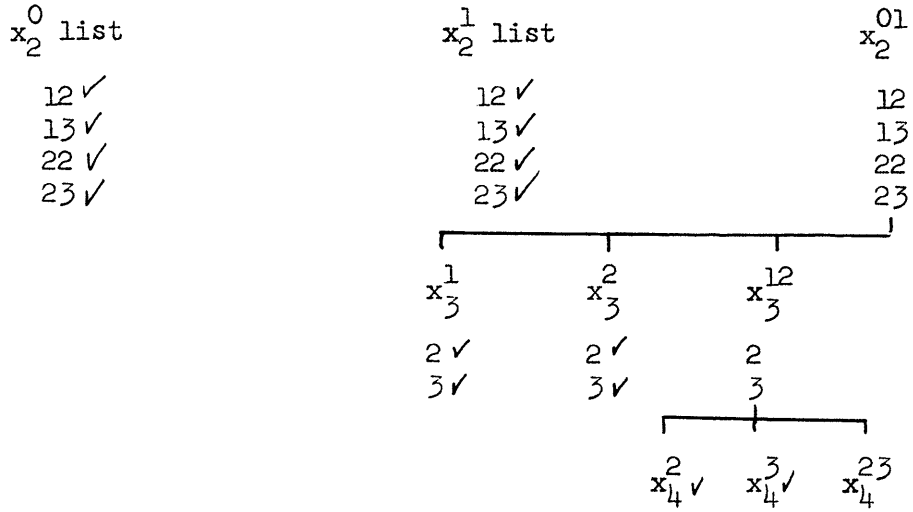
x_2^3 list

- 01 ✓
- 02 ✓
- 11 ✓
- 12 ✓

which lead to the following



Similar expansions on the variable x_2, x_3 of the x_1^{01} list lead to:



Tracing back we obtain the final form

$$\begin{aligned}
 & x_4^{23} x_3^{12} x_2^{01} x_1^{01} + x_4^{12} x_3^{01} x_2^{123} x_1^1 \\
 & + x_4^2 x_3^1 x_2^{0123} x_1^1 \\
 = & x_1^{01} x_2^{01} x_3^{12} x_4^{23} + x_1^1 x_2^{123} x_3^{01} x_4^{12} + x_1^1 x_3^1 x_4^2
 \end{aligned}$$

An effective way of handling the multiplicity of functions g_{m-1} through g_1 simultaneously is based on a modification of Polansky's "tag" method (Ref. 14). Here we preface each vertex (called kernel by Polansky) by a tag formed as follows: a tag has $m-1$ positions, labelled 1 through $m-1$ from right to left. A 0(1) is placed in position j if the tagged vertex must (must not) be in function g_j . If a 0 is placed in position j , then in positions 1 through $j-1$ a \emptyset is placed. All the true vertices of all g_i are placed on one list L_0 together with their tags. The list is processed in the manner of the previously given algorithm, except that tags are processed according to the following

rule: when terms derived from vertices are copied into a successor list, the new tag is formed digit-by-digit according to the following commutative rule:

	t'_j		
t_j^2	0	1	\emptyset
0	0	1	0
1	1	1	1
\emptyset	0	1	\emptyset

An item is checked iff the new tag equals the old tag.

To illustrate the procedure and illuminate some closing remarks, consider the following example:

	L_0	x_1^0	x_1^1	x_1^2	x_1^3
	TAG VERTEX	4 10 \emptyset 1 \checkmark	1 0 $\emptyset\emptyset$ 0	3 10 \emptyset 0 \checkmark	2 0 $\emptyset\emptyset$ 2
	$x_1 x_2$	9 110 2 \checkmark	5 10 \emptyset 1 \checkmark	8 110 1 \checkmark	7 10 \emptyset 3
			6 10 \emptyset 2	10 110 2 \checkmark	
1	0 $\emptyset\emptyset$	10		11 110 3 \checkmark	
2	0 $\emptyset\emptyset$	32			
3	10 \emptyset	20	x_1^{01}	x_1^{12}	x_1^{23}
4	10 \emptyset	01			
5	10 \emptyset	11	4,5 10 \emptyset 1	3 10 \emptyset 0	10 110 2 \checkmark
6	10 \emptyset	12	9 110 2	8 110 1 \checkmark	11 110 3
7	10 \emptyset	33		10 110 2 \checkmark	
8	110	21			
9	110	02	x_1^{012}	x_1^{123}	
10	110	22			
11	110	23	8 110 1	10 110 2 \checkmark	
			9,10 110 2 \checkmark		
			x_1^{0123}		
			9,10 110 2		

$$\begin{array}{c}
 x_1^1 \\
 x_2^0 \quad x_2^1 \quad x_2^2 \\
 1 \quad 0\emptyset\emptyset \quad 5 \quad 10\emptyset\checkmark \quad 6 \quad 10\emptyset\checkmark
 \end{array}$$

$$\begin{array}{c}
 x_2^{01} \quad x_2^{12} \\
 5 \quad 10\emptyset\checkmark \quad 5,6 \quad 10\emptyset\checkmark
 \end{array}$$

$$\begin{array}{c}
 x_2^{012} \\
 \Downarrow \\
 1 \quad x_1^1 x_2^0 \\
 5,6 \quad x_1^1 x_2^{012}
 \end{array}$$

$$\begin{array}{c}
 x_1^{01} \\
 x_2^1 \quad x_2^2 \\
 4,5 \quad 10\emptyset \quad 9 \quad 110
 \end{array}$$

$$\begin{array}{c}
 \Downarrow \\
 9 \quad x_1^{01} x_2^{12} \\
 4,5 \quad x_1^{01} x_2^1
 \end{array}$$

$$\begin{array}{c}
 x_1^{23} \\
 \Downarrow \\
 10,11 \quad x_1^{23} x_2^{23}
 \end{array}$$

$$\begin{array}{c}
 x_1^{0123} \\
 \Downarrow \\
 9,10 \quad x_1^{0123} x_2^2 = x_2^2
 \end{array}$$

$$\begin{array}{c}
 x_1^3 \\
 x_2^2 \quad x_2^3 \\
 2 \quad 0\emptyset\emptyset \quad 7 \quad 10\emptyset
 \end{array}$$

$$\begin{array}{c}
 \Downarrow \\
 7 \quad x_1^3 x_2^{23} \\
 2 \quad x_1^3 x_2^2
 \end{array}$$

$$\begin{array}{c}
 x_1^{12} \\
 x_2^0 \quad x_2^1 \quad x_2^2 \\
 3 \quad 10\emptyset \quad 8 \quad 100 \quad 10 \quad 110
 \end{array}$$

$$\begin{array}{c}
 \Downarrow \\
 3 \quad x_1^{12} x_2^0 \\
 8,10 \quad x_1^{12} x_2^{012}
 \end{array}$$

$$\begin{array}{c}
 x_1^{012} \\
 \Downarrow \\
 8,9,10 \quad x_1^{012} x_2^{12}
 \end{array}$$

The products obtained and the vertices which they cover can now be entered in a table, analogous to the conventional table of coverage of prime implicants. This is done below.

	1	2	3	4	5	6	7	8	9	10	11
E $x_1^1 x_2^0$	✓										
E $x_1^1 x_2^{012}$					✓	✓					
E $x_1^3 x_2^{(2)3}$							✓				
E $x_1^3 x_2^2$		✓									
$x_1^{01} x_2^{(1)2}$									✓		
E $x_1^{01} x_2^1$				✓	✓						
E $x_1^{(1)2} x_2^0$			✓								
$x_1^{(1)2} x_2^{012}$								✓		✓	
E $x_1^{23} x_2^{23}$										✓	✓
$x_1^{012} x_2^{12}$								✓	✓	✓	
x_2^2									✓	✓	

Next we can start to process the table in the conventional manner.

First, we single out those vertices which are covered by only one product and mark the corresponding products with E (essential) to show that a cover must contain that product (or one derived from it). In our example, we find that all vertices except 5, 8, 9, and 10 lead to

essential products. If our criterion of goodness is to minimize the total number of products, then vertices 5 and 10 are covered by 6 or 4 and 11, respectively, so that we can omit 5 and 10 from further consideration. Finally, the covers of 8 and 9 are obtained from the single product $x_1^{012} x_2^{12}$. This then completes the selection of a minimum set of products.

However, it is not at all certain that the criterion of minimizing the total number of products is the most reasonable one to use. Recall that the term $x_p^{i \dots j}$ requires two gates: $x_p^{01 \dots i \dots j}$ and $x_p^{i \dots j j+1 \dots m-1}$. Each of these gates requires a transistor circuit, and one may wish to minimize the total number of these gates. The minimization based on this criterion is a very difficult one and we have no good procedure for achieving it. For cases where the number of independent variables is small, good heuristics are not difficult to formulate. For the general case, however, the problem is very hard as might be expected from the realization that the problem is analogous to the minimum-inverter problem in two-valued logic. Of course, we realize that the term $x_p^{i(i+1) \dots j}$ is always more expensive than the term $x_p^{01 \dots i \dots j}$, and that $x_p^{i(i+1) \dots j}$ is more expensive than $x_p^{i(i+1) \dots j(j+1) \dots m-1}$. But other than that, we have had no important basic insights.

Finally, we want to point out that the modified Scheinman procedure does not give all the shortest products. In our example, the product x_1^2 is not obtained, although it covers vertices 8, 10, and 11. If one wanted to obtain it, he would have to expand around x_2 first and then around x_1 . In general, if all shortest products are to be obtained, all possible sequences of decomposition would have to be executed.

CHAPTER 3

SYMMETRY IN M-VALUED FUNCTIONS

Frequently in logical design the function to be synthesized is in some sense independent of the ordering of the inputs. For instance, the output of a summer depends on the values taken on by the inputs, but not on how these values are distributed among the inputs. This notion leads to the well-known definition of symmetric functions in bivalued switching.

Definition 3.1: A function f of n variables x_1, \dots, x_n is symmetric in $x_1^*, x_2^*, \dots, x_n^*$ (variable x_i^* being either x_i or \bar{x}_i) iff $f(x_1^*, \dots, x_n^*) = f(x_{p(1)}^*, \dots, x_{p(n)}^*)$ for all $n!$ permutations of the variables, p .

It is readily seen that this amounts to saying that any interchange of variables, say x_i^* with x_j^* , will leave the function invariant.

It can also be shown that in the bivalued case an equivalent definition is

Definition 3.2: A function f of n variables x_1, \dots, x_n is symmetric in $x_1^*, x_2^*, \dots, x_n^*$ iff f depends only on the real sum $\sum_{i=1}^n x_i^*$ of the transformed inputs.

It is this fact that enables one to express a symmetric function f as

$$f(x_1, \dots, x_n) = S_A(x_1^*, \dots, x_n^*)$$

with A being the set of values taken on by the sum $\sum_i x_i^*$ for which f is 1.

In this chapter, we generalize this notion to multilevel switching theory.

3.1 DEFINITIONS OF SYMMETRY IN M-LEVEL FUNCTIONS

Henceforth, unless otherwise stated, we shall always let m denote the number of truth values in the logic. By S_i will be meant the complete group of permutation of i objects (i.e., the symmetric group of degree i and order $i!$). Elements of S_n will be denoted p, p_1, p_2, \dots etc. S_i^k will denote the k^{th} Cartesian product of S_i . If x_i is an m -valued variable, we shall denote a permutation, say p , of its m truth values by the transformed variable x_i^p . When x_i takes on truth value k , x_i^p will take on value $p(k)$. We shall always denote the variables of a given function by x_1, \dots, x_n , and use y_1, \dots, y_n , or z_1, \dots, z_n to denote "transforms" of the x_i 's, i.e., y_i or z_i is a particular x_i^p .

Corresponding to Definition 3.1, we have the following

Definition 3.3: A function f of n variables x_1, \dots, x_n is 1-symmetric in y_1, \dots, y_n iff for all p in S_n $f(y_1, \dots, y_n) = f(y_{p(1)}, \dots, y_{p(n)})$.

Definition 3.4: A function $f(x_1, \dots, x_n)$ is 1-symmetric iff $(\exists p_1, \dots, p_n \in S_m) [f \text{ is 1-symmetric in } x_1^{p_1}, \dots, x_n^{p_n}]$.

It is readily shown that f is 1-symmetric in $x_1^{p_1}, \dots, x_n^{p_n}$ iff interchanging any two transformed variables, say $x_i^{p_i}$ and $x_j^{p_j}$, leaves the function invariant.

Corresponding to definition 3.2, we have a second definition of symmetry in m -level switching:

Definition 3.5: A function f of n variables x_1, \dots, x_n is 2-symmetric in y_1, \dots, y_n with the p_i 's, as before, being the transformations of the inputs, iff f depends only on the real sum $\sum_{i=1}^n y_i$ of the transformed variables.

Definition 3.6: A function $f(x_1, \dots, x_n)$ is 2-symmetric iff
 $(\exists p_1, \dots, p_n \in S_m)$ [f is 2-symmetric in $x_1^{p_1}, \dots, x_n^{p_n}$].

Since the sum $\sum_i x_i^{p_i}$ can take on any value from 0 to $n(m-1)$, corresponding to each of the $m^{n(m-1)+1}$ partitionings of $0, 1, \dots, n(m-1)$ into m blocks A_0, \dots, A_{m-1} , we can write a function f which is 2-symmetric in y_1, \dots, y_n as

$$f(x_1, \dots, x_n) = S_{A_1, \dots, A_{m-1}}(y_1, \dots, y_n)$$

with each A_i containing the value of $\sum_i y_i$ for which f takes on the value i . The omission of A_0 is made arbitrarily. If the sum of the transformed inputs is not found in any of the $m-1$ sets A_1, \dots, A_{m-1} , f is assumed to take on the value 0.

From a synthesis viewpoint, 2-symmetric functions lead directly to threshold logic realizations. (In fact it is well known in binary switching that most arithmetic units, which turn out to be symmetric functions, are easiest to implement by threshold devices.)

The transformed inputs are connected to a common linear summer with the set of weights $(1, 1, \dots, 1)$. In particular, the value function (see Ref. 16) of a 2-symmetric function

$$f(Y) = S_{A_1, A_2, \dots, A_{m-1}}(y_1, \dots, y_n)$$

is simply a string of $n(m-1)+1$ specified digits originating at the origin (*). The value of the i^{th} digit is j if i is in A_j and is 0 if i is not in A_1, \dots, A_{m-1} .

Intuitively one thinks of a function f as 1-symmetric in x_1, \dots, x_n if the variables are indistinguishable. In fact, if one has a black box with n input terminals that realizes f , he can connect x_1, \dots, x_n to the n input terminals in any way he likes without affecting the output.

From intuition, one would expect that such a function can be realized by a threshold-element (TE) network with all input weights equal. However, this turns out to be false, as we shall see.

In the next section, we shall discuss some of the properties of 1- and 2- symmetry. We will develop detection algorithms and prove that 2-symmetry implies 1-symmetry, but not vice-versa.

3.2 PROPERTIES OF SYMMETRIC FUNCTIONS

We shall establish certain properties of 1- and 2-symmetric functions so that effective algorithms can be derived to detect such functional behavior. Henceforth, an unqualified "symmetric" will mean both "1-symmetric" and "2-symmetric".

For a given function $f(x_1, \dots, x_n)$, we can group the m^n vertices into m groups (or truth sets) corresponding to the value of f . Conversely, any m groups of truth-sets uniquely defines a function f . If we arrange the vectors of the truth sets for $f = 0, 1, 2, \dots, m-1$ in a column (in that order), we obtain an $m^n \times n$ matrix. The top block of n -tuples, corresponding to the $f = 0$ vertices, can be discarded as being redundant. The remaining matrix, which uniquely specifies the function f , is called a principle matrix for f . Each of the $m-1$ blocks (some of which may be empty) of vertices corresponding to $f = 1, 2, \dots, m-1$ will be referred to as the submatrices of the principal matrix.

The common form of a principal matrix is

$$\begin{matrix} f=1 \\ \vdots \\ f=m-1 \end{matrix} \left[\begin{array}{c} \hline V_1 \\ \vdots \\ \hline V_{m-1} \end{array} \right]$$

Later it will be necessary to discuss the number of 0's, 1's, and so on in a column of a truth-table matrix or submatrix. These numbers will be referred to as the matrix or submatrix 0,1,...,m-1 counts.

Two columns of a submatrix are said to have equivalent column counts iff the 0,1,...,m-1 counts for one column can be made equal to the corresponding counts for the other column by a permutation $p \in S_m$ of the m counts. Two columns of a principal matrix are said to have compatible column counts iff in the (m-1) submatrices the column counts of the two columns are all equivalent via the same permutation p. Hence, if the column counts for the i^{th} and j^{th} columns are compatible via the permutation p, we can make the column counts in these columns identical in each submatrix by transforming the appropriate variable by p.

Generalizing, if all n columns of a principal matrix have compatible column counts*, it is possible to transform variables x_2, \dots, x_n by the appropriate $p_2, \dots, p_n \in S_m$ so that the resulting principal matrix, called a standard matrix has in each of its (m-1) submatrices identical column counts for each column. For example, the function f specified by the principal matrix in Table 3.1(a) has equivalent column counts in both submatrices. Furthermore, the counts for the two columns are compatible via $p = (021)$. Hence f has a standard submatrix in $x_1 x_2^p$ as shown in Table 3.1(b).

Clearly if f has a standard matrix in terms of the variables y_1, \dots, y_n , f must also have standard matrices in terms of y_1^p, \dots, y_n^p ,

* Since the relationship of being compatible is transitive, it makes sense to speak of more than two columns being compatible.

	x_1	x_2		x_1	x_2^p
$f = 1$	0	0		0	2
	0	2		0	1
	1	1		1	0
	1	2		1	1
	2	1		2	0
0 count	2	1	0 count	2	2
1 count	2	2	1 count	2	2
2 count	1	2	2 count	1	1
$f = 2$	0	1		0	0
	2	0		2	2
0 count	1	1	0 count	1	1
1 count	0	1	1 count	0	0
2 count	1	0	2 count	1	1

(a)

$p = (021)$
(b)

Table 3.1 (a) Principal Matrix for f

(b) Standard Matrix of f in Terms of x_1, x_2^p

where p is any one permutation of S_m . Of course, some of these standard matrices may well be the same.

Another convenient parameter for a principal matrix is the sum of the truth values in each row. This sum for the matrix row corresponding to vertex $V_i = (x_{1i}, x_{2i}, \dots, x_{ni})$ is

$$r = \sum_{j=1}^n x_{ji}$$

is called the row weight for V_i . The row weight and column counts are the basic parameters used to determine whether the function is symmetric.

Given the principal matrix of a function, quite obviously it would be possible to determine if the function is symmetric by examining all

$(m!)^n$ possible transforms $x_1^{p_1}, \dots, x_n^{p_n}$ of x_1, \dots, x_n . We shall see that a much simpler approach exists for detecting symmetry.

Lemma 3.1: $S_{A_1, \dots, A_{m-1}}(y_1, \dots, y_i, \dots, y_j, \dots, y_n) = S_{A_1, \dots, A_{m-1}}(y_1, \dots, y_j, \dots, y_i, \dots, y_n)$.

Proof: The truth value of a 2-symmetric function depends on the real sum of the inputs. The lemma is a consequence of the commutativity of summation.

Hence, we have shown that f is 2-symmetric in $x_1^{p_1}, \dots, x_n^{p_n}$ only if f is 1-symmetric in $x_1^{p_1}, \dots, x_n^{p_n}$. The converse, however, is not true. The function shown in Table 3.2 is clearly 1-symmetric in x_1, x_2 . However, it is not 2-symmetric in x_1, x_2 since when $x_1 + x_2 = 2$, f is not defined uniquely. Note, however, that f is 2-symmetric in x_1^p, x_2^p with p being the transformation (01)(2).

	x_1	x_2		x_1^p	x_2^p			
$f = 1$	[1	1]	2	[0	0]	0
$f = 2$	[0	2]	2	[1	2]	3
	[2	0]	2	[2	1]	3
		(a)			$p = (01)(2)$			
					(b)			

Table 3.2 (a) f is 1-Symmetric But Not 2-Symmetric in x_1, x_2

(b) f is 1-Symmetric and 2-Symmetric in x_1^p, x_2^p

Theorem 3.1: A function f is 1-symmetric only if it has a standard matrix in some set of transformed variables.

Proof: By definition 3.3, f is 1-symmetric in y_1, \dots, y_n only if interchanging any columns in the principal matrix in terms of y_1, \dots, y_n

leaves the functional description invariant. For the functional description to be invariant, the most that can happen to the matrix is a reordering of the rows in each submatrix. Since the latter leaves the column counts in each column undisturbed, whereas interchanging columns interchanges the column counts of the two columns, it follows that the column counts for the n columns must be identical in each submatrix, i.e., that the principal matrix is a standard matrix.

By Lemma 3.1, which implies that f is 2-symmetry only if it is 1-symmetric, we have the trivial

Corollary: A function f is 2-symmetric only if it has a standard matrix in terms of some set of transformed variables.

We have shown a necessary condition for 2-symmetry. The next theorem will supply a necessary and sufficient condition for 1-symmetry.

Theorem 3.2: A function f with a principal matrix in terms of y_1, \dots, y_n is 1-symmetric in y_1, \dots, y_n iff in each submatrix of the standard matrix every row vector is accompanied by all the other vectors which differs from it only in the ordering of the row elements.

Proof: If f is 1-symmetric in $x_1^{p_1}, \dots, x_n^{p_n}$ and yet in one submatrix the row vector $(x_{1i}, x_{2i}, \dots, x_{ni})$ appears unaccompanied by $(x_{2i}, x_{1i}, \dots, x_{ni})$, and $x_{1i} \neq x_{2i}$, then $f(y_1, y_2, \dots, y_n)$, will not be the same function as $f(y_2, y_1, \dots, y_n)$, contrary to the assumption that f is 1-symmetric in the transformed variables.

On the other hand, if in the principal matrix every submatrix has each of its vectors accompanied by all of its permuted relatives, then any permutation of the columns will leave the vectors of each submatrix invariant. Hence the function remains the same, i.e., is 1-symmetric

in the transformed variables. Note that we have shown that the principal matrix must also be a standard matrix.

Suppose the principal matrix of a function f satisfies the requirements of Theorem 3.2. There is still no assurance at this point that f is 2-symmetric. For 2-symmetry, the basic question that arises is this: in each submatrix, if a vector (row) has a row weight r , then is each vector in A^n with row weight r also in the submatrix? That is, do each of the standard submatrices have a proper row-weight distribution?

Clearly, f is 2-symmetric in the transformed variables only if each of the standard submatrices has a proper row weight distribution. For otherwise, there will be two vertices (a_1, \dots, a_n) and (b_1, \dots, b_n) with $\sum p_i(a_i) = \sum p_i(b_i)$ and yet $f(a_1, \dots, a_n) \neq f(b_1, \dots, b_n)$. Hence, f will not depend only on the real sum $\sum_i x_i^{p_i}$ of the transformed variables and will not be 2-symmetric. On the other hand, if each standard submatrix does have a proper row-weight distribution, f is indeed 2-symmetric in the variables $x_1^{p_1}, \dots, x_n^{p_n}$ of that particular standard matrix. In fact, if for $i = 1, 2, \dots, m-1$, A_i is the set of distinct values taken on by the row-weights in the i^{th} submatrix, proper row-weight distribution in each submatrix guarantees the sets A_i to be mutually exclusive. Furthermore, f is readily seen to be the 2-symmetric function $S_{A_1, \dots, A_{m-1}}(x_1^{p_1}, \dots, x_n^{p_n})$. This can be stated in the following

Theorem 3.3: A function $f(x)$ is 2-symmetric iff there is a transformation of the variables in terms of which f has a standard matrix with a proper row-weight distribution.

In bivalued logic with truth values 0,1, the number occurrences of combinations of n variables having a linear sum r is given by the

coefficient of the term x^r in the expansion of $(1+x)^n$. This is the well-known binomial coefficient $C(n,r)$. In m -valued logic, each variable may take on the value $0,1,\dots,m-1$ - leading to the generating function $(1+x+x^2+\dots+x^{m-1})^n$. Hence, we have the following

Lemma 3.2: The number of vectors in A^n with a row weight r is given by the coefficient of the term x^r in the expansion of

$$(1+x+x^2+\dots+x^{m-1})^n$$

There are two ways to express this coefficient, which we denote by $N_m^n(r)$. The customary form for $N_m^n(r)$ is obtained by a simple application of the Multinomial Theorem which yields

$$N_m^n(r) = \sum_{(i)} \frac{n!}{i_0! i_1! \dots i_{m-1}!}$$

where the sum is over all m -tuples of non-negative integers satisfying

$$\begin{aligned} \text{i)} \quad & \sum_{k=0}^{m-1} i_k = n \\ \text{ii)} \quad & \sum_{k=0}^{m-1} ki_k = r \end{aligned}$$

Instead of searching for all the partitions (i), one may alternatively calculate the coefficient of x^r by observing that

$$(1+x+x^2+\dots+x^{m-1})^n = (1-x^m)^n (1-x)^{-n}$$

Hence,

$$N_m^n(r) = \sum_{i=0}^{\lfloor r/m \rfloor} (-1)^i \frac{n(n+r-mi-1)!}{i! (n-i)! (r-mi)!}$$

This form for $N_m^r(r)$ often leads to more straight-forward calculations.

Briefly then, it is known from Theorem 3.1 that a function f is 2-symmetric only if it has a standard matrix. By Theorem 3.3, f is

2-symmetric on the variables of the standard matrix only if the standard submatrices have the proper row-weight distributions. Lemma 3.2 provides an analytical means of determining whether a proper row-weight distribution exists.

We come then to the question of whether a standard matrix for $f(x)$ is a proper standard matrix, that is, a matrix with submatrices having proper row-weight distributions. Consider first the case where $f(x)$ has a standard matrix in terms of y_1, \dots, y_n and every pair of column counts are distinct in at least one submatrix. Such a function cannot have a standard matrix in terms of $y_1^{p_1}, \dots, y_n^{p_n}$ with some $p_i \neq p_j$ as is easily shown. Therefore, f can have at most $m!$ distinct standard matrices corresponding to the $m!$ permutations of S_m .

By Theorem 3.3 it is sufficient to test all these for proper row-weight distribution and if at least one standard matrix is proper, as is the case shown in Table 3.3, f is 2-symmetric. If none of them is found to be proper, as in the case of the example shown in Table 3.4, then f is not 2-symmetric. Note, incidentally, that f is also not 1-symmetric - a necessary condition for 2-symmetry. (Later, an example will be given of a function f with standard matrices and f will be shown to be 1-symmetric but not 2-symmetric.)

One property of S_m will enable us to decrease the number of standard matrices that need be examined from $m!$ to $m!/2$. Consider the element p of S_m that reverses the ordering in A , i.e., $x^p = m-1-x$. We shall denote x^p by \bar{x} .

Lemma 3.3: If f is 2-symmetric in the transformed variables y_1, \dots, y_n , then f is also 2-symmetric in the variables $\bar{y}_1, \dots, \bar{y}_n$.

	x_1	x_2		$\overset{p_1}{x_1}$	$\overset{p_1}{x_2}$		$\overset{p_2}{x_1}$	$\overset{p_2}{x_2}$	
$f = 1$	$\begin{pmatrix} 0 \\ 1 \\ 2 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 2 \\ 1 \\ 1 \end{pmatrix}$	0 3 3 2	$\begin{pmatrix} 1 \\ 0 \\ 2 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 2 \\ 0 \\ 0 \end{pmatrix}$	2 2 2 0	$\begin{pmatrix} 0 \\ 2 \\ 1 \\ 2 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 1 \\ 2 \\ 2 \end{pmatrix}$	0 3 3 4
0 count	1	1		2	2		1	1	
1 count	2	2		1	1		1	1	
2 count	1	1		1	1		2	2	
$f = 2$	$\begin{pmatrix} 2 \\ 0 \\ 2 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 2 \\ 2 \end{pmatrix}$	2 2 4	$\begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix}$	3 3 4	$\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$	1 1 2
0 count	1	1		0	0		1	1	
1 count	0	0		1	1		2	2	
2 count	2	2		2	2		0	0	
	(a)			$p_1 = (01)(2)$			$p_2 = (0)(12)$		
				(b)			(c)		

Table 3.3 (a) Standard Matrix of f with Improper Row Weight Distribution
 (b) Standard Matrix of f with Proper Row Weight Distribution
 (c) Standard Matrix of f with Improper Row Weight Distribution

Proof: If f is 2-symmetric in y_1, \dots, y_n , it can be written as $S_{A_1, \dots, A_{m-1}}(y_1, \dots, y_n)$, i.e., f has a proper standard matrix in terms of y_1, \dots, y_n . Consider the principal matrix of f in terms of $\bar{y}_1, \dots, \bar{y}_n$. As noted before, this matrix must also be a standard matrix. By Theorem 3.3, it is sufficient to show that this standard matrix is also proper.

First observe that any row in the standard matrix in terms of y_1, \dots, y_n that sums to p will in the new matrix sum to $n(m-1)-p$. Since the original matrix is proper, all that we have to show is that $N_m^n(p) = N_m^n(n(m-1)-p)$. Using the notions of generating functions, we see that another way of obtaining $N_m^n(n(m-1)-p)$ is by finding the coefficient of the term y^p in the function obtained by replacing each x^i by y^{m-1-i} for

	x_1	x_2		p_1 x_1	p_1 x_2		p_2 x_1	p_2 x_2	
$f = 1$	$\begin{pmatrix} 0 \\ 1 \\ 2 \\ 2 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 2 \\ 0 \\ 2 \end{pmatrix}$	$\begin{matrix} 1 \\ 3 \\ 2 \\ 4 \end{matrix}$	$\begin{pmatrix} 1 \\ 0 \\ 2 \\ 2 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 2 \\ 1 \\ 2 \end{pmatrix}$	$\begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix}$	$\begin{pmatrix} 2 \\ 1 \\ 0 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 0 \\ 2 \\ 0 \end{pmatrix}$	$\begin{matrix} 3 \\ 1 \\ 2 \\ 0 \end{matrix}$
0 count	1	1		1	1		2	2	
1 count	1	1		1	1		1	1	
2 count	2	2		2	2		1	1	
$f = 2$	$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 2 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 2 \\ 1 \end{pmatrix}$	$\begin{matrix} 1 \\ 0 \\ 2 \\ 3 \end{matrix}$	$\begin{pmatrix} 0 \\ 1 \\ 1 \\ 2 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \\ 2 \\ 0 \end{pmatrix}$	$\begin{matrix} 1 \\ 2 \\ 3 \\ 2 \end{matrix}$	$\begin{pmatrix} 1 \\ 2 \\ 2 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 2 \\ 2 \\ 0 \\ 1 \end{pmatrix}$	$\begin{matrix} 3 \\ 4 \\ 2 \\ 1 \end{matrix}$
0 count	2	2		1	1		1	1	
1 count	1	1		2	2		1	1	
2 count	1	1		1	1		2	2	
	$p_0 = (0)(1)(2)$			$p_1 = (01)(2)$			$p_2 = (02)(1)$		
	(a)			(b)			(c)		
	p_3 x_1	p_3 x_2		p_4 x_1	p_4 x_2		p_5 x_1	p_5 x_2	
0 count	$\begin{pmatrix} 0 \\ 2 \\ 1 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 2 \\ 1 \\ 0 \\ 1 \end{pmatrix}$	$\begin{matrix} 2 \\ 3 \\ 1 \\ 2 \end{matrix}$	$\begin{pmatrix} 1 \\ 2 \\ 0 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 2 \\ 0 \\ 1 \\ 0 \end{pmatrix}$	$\begin{matrix} 3 \\ 2 \\ 1 \\ 0 \end{matrix}$	$\begin{pmatrix} 2 \\ 0 \\ 1 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 1 \\ 2 \\ 1 \end{pmatrix}$	$\begin{matrix} 2 \\ 1 \\ 3 \\ 2 \end{matrix}$
1 count	2	2		1	1		2	2	
2 count	1	1		1	1		1	1	
0 count	$\begin{pmatrix} 2 \\ 0 \\ 0 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 1 \\ 2 \end{pmatrix}$	$\begin{matrix} 2 \\ 0 \\ 1 \\ 3 \end{matrix}$	$\begin{pmatrix} 2 \\ 1 \\ 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \\ 0 \\ 2 \end{pmatrix}$	$\begin{matrix} 3 \\ 2 \\ 1 \\ 2 \end{matrix}$	$\begin{pmatrix} 0 \\ 2 \\ 2 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 2 \\ 2 \\ 1 \\ 0 \end{pmatrix}$	$\begin{matrix} 2 \\ 4 \\ 3 \\ 1 \end{matrix}$
1 count	1	1		2	2		1	1	
2 count	1	1		1	1		2	2	
	$p_3 = (12)(0)$			$p_4 = (012)$			$p_5 = (021)$		
	(d)			(e)			(f)		

Table 3.4 A Function with Standard Matrices which Is Neither 2-Symmetric Nor 1-Symmetric

$i = 0, 1, \dots, m-1$ in the generating function of Lemma 3.2. But the new function is $(y^{m-1} + y^{m-2} + \dots + y + 1)^n$ and is the same function as before. This shows that the new standard matrix is indeed proper and concludes the proof.

Note that since $\bar{\bar{x}} = x$, the converse of the lemma is also true. Hence whenever one standard matrix of f in terms of y_1, \dots, y_n is proven not to be proper, we need not examine the matrix in terms of $\bar{y}_1, \dots, \bar{y}_n$.

We now treat the case where exactly $q (q \geq 2)$ column counts are non-distinct in every standard submatrix. In this case we can have as many as $(m!)(q!)^{n-1}$ different standard matrices for f , as is readily verified. Rather than examining these exhaustively, it is often simpler to decompose the function about one of its transformed variables, say y_1 , obtaining the m functions of $n-1$ variables $f(0, y_2, \dots, y_n), \dots, f(i, y_2, \dots, y_n), \dots, f(m-1, y_2, \dots, y_n)$. The conditions that these functions must satisfy to imply f to be 2-symmetric will be supplied by the following

Theorem 3.4: A 2-symmetric function $S_{A_1, \dots, A_{m-1}}(Y)$ can be expanded about any variable, say y_1 , to obtain the following m residues:

$$S_{A_1, \dots, A_{m-1}}(Y) = \begin{cases} S_{A_1^0, \dots, A_{m-1}^0}(y_2, \dots, y_n) & \text{if } y_1 = 0 \\ \vdots \\ S_{A_1^i, \dots, A_{m-1}^i}(y_2, \dots, y_n) & \text{if } y_1 = i \\ \vdots \\ S_{A_1^{m-1}, \dots, A_{m-1}^{m-1}}(y_2, \dots, y_n) & \text{if } y_1 = m-1 \end{cases}$$

where each new set A_j^i is formed from A_j by subtracting i from every member, excluding all those results that are less than 0.

Proof: If $y_1 = i, 0 \leq i \leq m-1$, then by the definition of the A_i^j 's, whenever $\sum_{k=1}^n y_k$ is in $A_j^i, 0 \leq p \leq m-1, \sum_{k=2}^n y_k$ will be in A_j^i . This is all that the theorem claims. Furthermore, if a number less than i is in any A_p , then for $y_1 = i$ it is impossible for $\sum_{k=1}^n y_k$ to equal that number. This justifies our restriction.

The following algorithm, based on Theorem 3.4, can be employed whenever a standard matrix of f in terms of y_1, \dots, y_n has degenerate column counts. We shall denote $f(i, y_2, \dots, y_n)$ as g_i , for $i = 0, 1, \dots, m-1$.

Step 1: Form a principal matrix for $g_0 = f(0, y_2, \dots, y_n)$. By methods discussed earlier, obtain a proper standard matrix for g_0 . If this cannot be done, g_0 is not 2-symmetric and so, by Theorem 3.4, neither is f . If a degenerate and non-proper standard matrix is obtained, then this algorithm has to be applied to g_0 (note the recursion). Otherwise, $g_0 = S_{B_1^0, \dots, B_{m-1}^0} (z_2, \dots, z_n)$.

Step 2: For $i = 1, \dots, m-1$, obtain each g_i as $S_{B_1^i, \dots, B_{m-1}^i} (z_2, \dots, z_n)$. If this is not possible, then f is not 2-symmetric in the transformed variables y_1^p, z_2, \dots, z_n for any p . Go back to Step 1 and see if another set of transforms Z of Y can be obtained such that $g_i = S_{B_1^i, \dots, B_{m-1}^i} (z_2, \dots, z_n)$, for $i = 0, 1, \dots, m-1$. The sets B_i^j obtained will be different from the previous sets B_i^j . (Since most functions would only be 2-symmetric in a very small number of sets of transformed variables, we can be sure that this repetition is of no major importance.) If no transformed variables z can be found, then, by Theorem 3.4, f cannot be 2-symmetric.

Step 3: At this point, we have a set of variables $(z_2, \dots, z_n) = (y_2^{p_2}, \dots, y_n^{p_n})$ such that

$$** \quad g_i = S_{B_1^i, \dots, B_{m-1}^i} (z_2, \dots, z_n) \quad i = 0, 1, \dots, m-1$$

Form an $m \times (m-1)$ array as follows:

$$\begin{array}{ccccccc} A_1^{p(0)} & = & B_1^0 & \dots & A_k^{p(0)} & = & B_k^0 & \dots & A_{m-1}^{p(0)} & = & B_{m-1}^0 \\ & & \vdots & & & & \vdots & & & & \vdots \\ A_1^{p(q)} & = & B_1^q & \dots & A_k^{p(q)} & = & B_k^q & \dots & A_{m-1}^{p(q)} & = & B_{m-1}^q \\ & & \vdots & & & & \vdots & & & & \vdots \\ A_1^{p(m-1)} & = & B_1^{m-1} & \dots & A_k^{p(m-1)} & = & B_k^{m-1} & \dots & A_{m-1}^{p(m-1)} & = & B_{m-1}^{m-1} \end{array}$$

The procedure described below will be repeated for all p in S_m such that the principal matrix of f in terms of y_1^p, z_2, \dots, z_n is a standard matrix.

Column k of the array is said to be consistent iff for each j , $0 \leq j \leq m-1$ the following is true: For all q , $0 \leq q \leq m-1$, any element b_q of B_k^q is matched by an element $b_j = b_q + p(q) - p(j)$ of B_k^j - unless b_j is less than 0 or greater than $(n-1)(m-1)$.

A permutation p leading to an array in which all $(m-1)$ columns are consistent is said to be an acceptable permutation.

From the construction, it is readily shown that if there is an acceptable permutation, p , f will be 2-symmetric in y_1^p, z_2, \dots, z_n . In

fact, if $A_k = \bigcup_{q=0}^{m-1} \{b_q \in B_k^q\}$, for $k = 1, \dots, m-1$

$$f(x_1, \dots, x_n) = S_{A_1, \dots, A_{m-1}} (y_1^p, z_2, \dots, z_n)$$

On the other hand, if no acceptable permutation p is found, we return to Step 1 and repeat until we exhaust all possible (z_2, \dots, z_n) 's

satisfying **. If still no acceptable permutation p can be found, we know, by Theorem 3.4, that f is not 2-symmetric.

To illustrate the above procedure, consider the degenerate, non-proper standard matrix shown in Table 3.5(a). f is seen to have $m!(q!)^{n-1} = 6 \cdot 2^2 = 24$ standard functions. Table 3.5(b) shows the standard matrix for g_0 obtained directly from (a). (b) is also a degenerate, non-proper, standard matrix. However, instead of decomposing about y_2 as the procedure suggests, we observe here that transforming y_2 by $p_2 = (02)(1)$ and y_3 by $p_3 = (012)$ leads to the proper standard matrix shown in Table 3.5(c). Here $(z_2, z_3) = (y_2^{p_2}, y_3^{p_3})$, and $g_0 = S_{(1,4)(0)}(z_2, z_3)$. From (a), we obtain immediately $g_1 = S_{(0,3)(4)}(z_2, z_3)$ and $g_2 = S_{(2)(0,1)}(z_2, z_3)$. Hence (z_2, z_3) satisfies the equation **. To check for acceptable permutations for y_1 , we note from the column counts of Table 3.5(a) that there are only two candidates: Either $p_3 = (012)$ or $p_1 = (02)(1)$. Let us try the second choice for the moment. Step 3 tells us to decide acceptability of p_1 by considering consistencies of the columns of the following array:

$$\begin{array}{ll} A_1^2 = (1,4) & A_2^2 = (0) \\ A_1^1 = (0,3) & A_2^1 = (4) \\ A_1^0 = (2) & A_2^0 = (01) \end{array}$$

A quick calculation shows that although the first column is consistent, the second one is not, since 0 in A_2^2 implies that a 1 be in A_2^1 and a 2 in A_2^0 .

Turning now to the other alternative $p_3 = (012)$, we form the array:

	y_1	y_2	y_3	
	1	0	0	1
	0	0	1	1
	1	1	1	3
	2	1	0	3
	0	2	0	2
	0	1	2	3
	2	2	1	5
	2	0	2	4
	1	2	2	5
0 count	3	3	3	
1 count	3	3	3	
2 count	3	3	3	
	2	2	2	6
	2	2	0	4
	2	1	2	5
	0	2	2	4
	1	0	1	2
0 count	1	1	1	
1 count	1	1	1	
2 count	3	3	3	

(a) Standard Matrix for f

	y_2	y_3	
	0	1	1
	2	0	2
	1	2	3
0 count	1	1	
1 count	1	1	
2 count	1	1	
	2	2	4
0 count	0	0	
1 count	0	0	
2 count	1	1	

(b) Matrix for g_0

	z_2	z_3	
	2	2	4
	0	1	1
	1	0	1
	0	0	0

(c) Proper Standard Matrix for g_0

	z_2	z_3	
	2	1	3
	1	2	3
	0	0	0
	2	2	4

(d) Proper Standard Matrix for g_1

	z_2	z_3	
	1	1	2
	0	2	2
	2	0	2
	0	0	0
	0	1	1
	1	0	1

(e) Proper Standard Matrix for g_2

Table 3.5 Degenerate Column Counts in Standard Matrix for f

$$\begin{array}{ll} A_1^1 = (1,4) & A_2^1 = (0) \\ A_1^2 = (0,3) & A_2^2 = (4) \\ A_1^0 = (2) & A_2^0 = (01) \end{array}$$

Both columns are consistent and hence p_3 is acceptable. Furthermore, $A_1 = (2,5) \quad (2,5) \quad (2) = (2,5)$, $A_2 = (1) \quad (6) \quad (0,1) = (0,1,6)$ and we conclude:

$$f = S_{(2,5)(0,1,6)} (y_1^{p_3}, y_2^{p_2}, y_3^{p_3})$$

We now have all the tools we need to generate a straight-forward procedure for the detection of 1- and 2-symmetry. Before we give the algorithm, the assertion made earlier after Lemma 3.1 will be proven in the following

Theorem 3.5: There exist 1-symmetric functions that are not 2-symmetric in any of its $(m!)^n$ sets of transformed variables, for $m \geq 5$.

We shall prove this theorem by explicitly showing one such function. Before we proceed, however, recall that the example of Table 3.2 is a function which is 1-symmetric in more sets of transformed variables than it is 2-symmetric. Such a function is by definition both 1-symmetric and 2-symmetric. The example in Table 3.4, on the other hand, is a function which is neither 1-symmetric nor 2-symmetric. What we want here is a function which, in terms of some transform of its variables, is invariant under any reordering of its inputs. On the other hand, we wish to show that such a function cannot be realized by a common-weight feed-forward network of threshold elements with n equal weights assigned to the transformed variables. For in such a network, the only

way the inputs can vary the output is by the value of their sum. We do so as follows:

Proof of Theorem 3.5: Consider the 5-valued function shown in Table 3.7. By Theorem 3.2, f is 1-symmetric in x_1x_2 , hence is 1-symmetric. However, consider what is necessary for f to be 2-symmetric. The

	x_1	x_2	
$f = 1$	$\left[\begin{array}{c} 0 \\ 4 \end{array} \right]$	$\left[\begin{array}{c} 4 \\ 0 \end{array} \right]$	4 4
	1	1	
	0	0	
	0	0	
	1	1	
$f = 2$	$\left[\begin{array}{c} 1 \\ 3 \end{array} \right]$	$\left[\begin{array}{c} 3 \\ 1 \end{array} \right]$	4 4
	0	0	
	1	1	
	0	0	
	1	1	
	0	0	
$f = 3$	$\left[\begin{array}{c} 2 \\ 0 \end{array} \right]$	$\left[\begin{array}{c} 2 \\ 0 \end{array} \right]$	4
	0	0	
	0	0	
	1	1	
	0	0	
	0	0	
$f = 4$	$\left[\begin{array}{c} \\ \end{array} \right]$	$\left[\begin{array}{c} \\ \end{array} \right]$	
	0	0	
	0	0	
	0	0	
	0	0	
	0	0	

Table 3.7

function has two truth sets with two vertices in each and one truth set with only one vertex. Clearly, in order for such a function to have a proper row weight distribution in each submatrix, the row sums can only be 0, 1, 7, or 8. For only

$$N_5^2(0) = N_5^2(8) = 1$$

and
$$N_5^2(1) = N_5^2(7) = 2$$

All other $1 < p < 7$ have $N_5^2(p) > 2$. Hence if f is 2-symmetric at all, in the variables in which it is 2-symmetric, f must choose five out of the following six combinations:

	$y_1 y_2$	Σ
1)	00	0
2)	44	8
3)	01	1
4)	10	1
5)	34	7
6)	43	7

In fact, choices 3), 4), 5), and 6) must be used for the two submatrices with two entries, and one of the remaining two has to be the entry in the $f = 3$ submatrix. However, in the original x_1 column of Table 3.7, truth values 0,1,2,3,4 occur exactly once each. Hence in any transformation of x_1 , they must also appear exactly once. Since both choices we are allowed to use to fill in the $f = 3$ submatrix will result in the transformed x_1 column having either two 0's or two 4's, we arrive at a contradiction. Hence f is not 2-symmetric.

Note that the same type of construction can be used for $m > 5$. However, it fails for $m \leq 4$. In fact, it is strongly suspected that the two classes of symmetric functions are the same for $m \leq 4$.

3.3 ALGORITHM TO DETECT SYMMETRIC FUNCTIONS

Using the properties established in Section 3.2, we shall describe an algorithm which, given any principal matrix of an m -valued function f , will determine if f is 1-symmetric or 2-symmetric.

Step 1: Form column counts for each submatrix of the principal matrix. If the condition of Theorem 3.1 is satisfied and f has a standard matrix in terms of some transform of its variables, then form the standard matrix and proceed to Step 2; otherwise, by Theorem 3.1, f is not symmetric.

Step 2: (If one is not concerned with 1-symmetry and is only interested in 2-symmetry, Step 2 can be skipped.) If the standard matrix satisfies the condition of Theorem 3.2, then f is 1-symmetric. If not, other standard matrices of f will have to be checked. If none of them is found to satisfy the condition of Theorem 3.2, f is not 1-symmetric, hence cannot be 2-symmetric.

Step 3: Next, the row weights are formed and checked for proper distribution. If they are found to be proper, by Theorem 3.3, f is 2-symmetric, and hence, by Lemma 3.1, 1-symmetric also. If the standard matrix does not have a proper row-weight distribution, and every pair of column counts is distinct in at least one submatrix, then go to Step 4. Otherwise, go to Step 5.

Step 4: By Theorem 3.3 and Lemma 3.3, it suffices in this case to check for proper-row weight distribution in the $m!/2$ non-equivalent (in the sense of Lemma 3.3) standard matrices of f . If all tests fail, then f is not 2-symmetric by Theorem 3.3.

Step 5: We have here a non-proper standard matrix with degenerate column counts. The algorithm described in the previous section following Theorem 3.4 can be applied to this matrix. Alternatively, if $(m!)(q!)^{n-1}$ is not too large, one may prefer to check all possible standard matrices instead.

3.4 SYNTHESIS OF SYMMETRIC FUNCTIONS

As mentioned earlier, 2-symmetric functions have a "natural" common-weight TE network realization. An m-valued full adder is a good example.

For two-valued switching, threshold elements are conveniently used for performing a digit-by-digit serial full add as shown in Figure 3.1. Here we develop a threshold network to realize a digit-by-digit full adder for m-valued signals. The algorithm presented in 3.3 will be used to obtain a realization.

First we note that for two-input adders the carry signal will never exceed 1. Hence the highest excitation achieved by the two inputs and the carry is $2m-1$. Clearly,

$$C_i = S_{m, m+1, \dots, 2m-1, \phi, \phi, \dots, \phi} (A_i, B_i, C_{i-1})$$

and the value function of C_i with all weights equal to one is simply

$$\begin{array}{c} * \\ \underbrace{000\dots0}_{m-1} \quad \underbrace{11\dots1}_m \text{-----} \end{array}$$

The STE realization of C_i in Figure 3.2 is obvious, where the first threshold is placed at $m-\frac{1}{2}$ and all others at arbitrary values higher than $2m-1$.

Similarly, the value function of the i^{th} sum S_i is easily seen to be

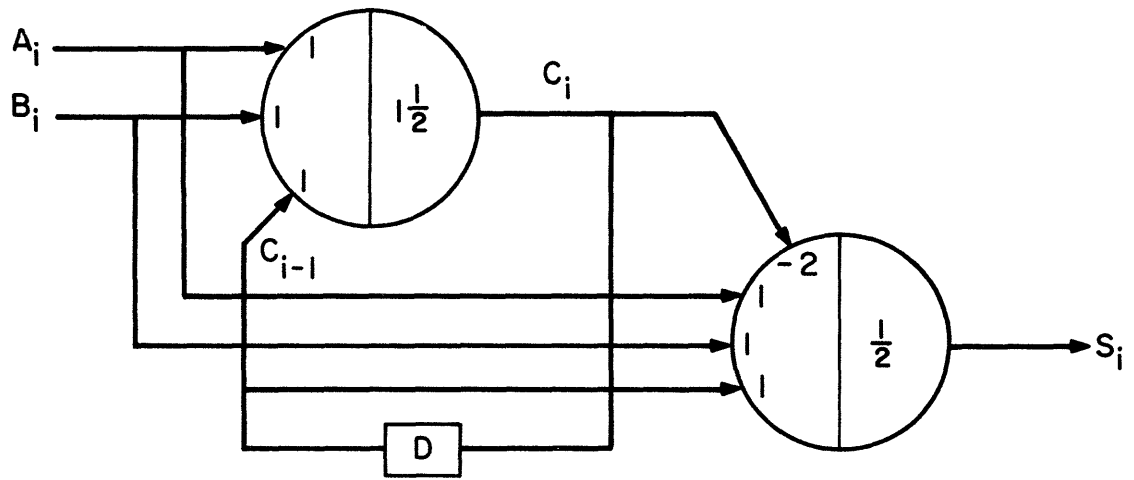


Fig. 3.1 Binary Full Adder

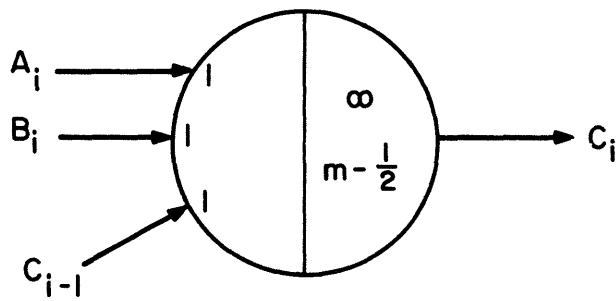


Fig. 3.2 m -Level STE Realization of C_i

$$*012\dots m-1 \ 012\dots m-1$$

The Sheng plot (Ref. 17) in Figure 3.3(a) indicates that a single "correction threshold" will be necessary and sufficient, i.e., for $E > m-1$, all thresholds should be increased by m . The realization of S_i is shown in Figure 3.3(b), where the upper element generates the correction threshold. One observes, however, that the upper element realizes exactly the bivalued carry function, C_i . Hence the circuit in Figure 3.4 is a digit-by-digit serial full adder for two m -ary numbers A and B .

If, instead of two, several m -ary numbers are to be summer (serially) simultaneously, the i^{th} carry will be able to take on more than one value. A little thought will show that the two threshold-element network of Figure 3.5 can sum up to m m -ary numbers simultaneously.

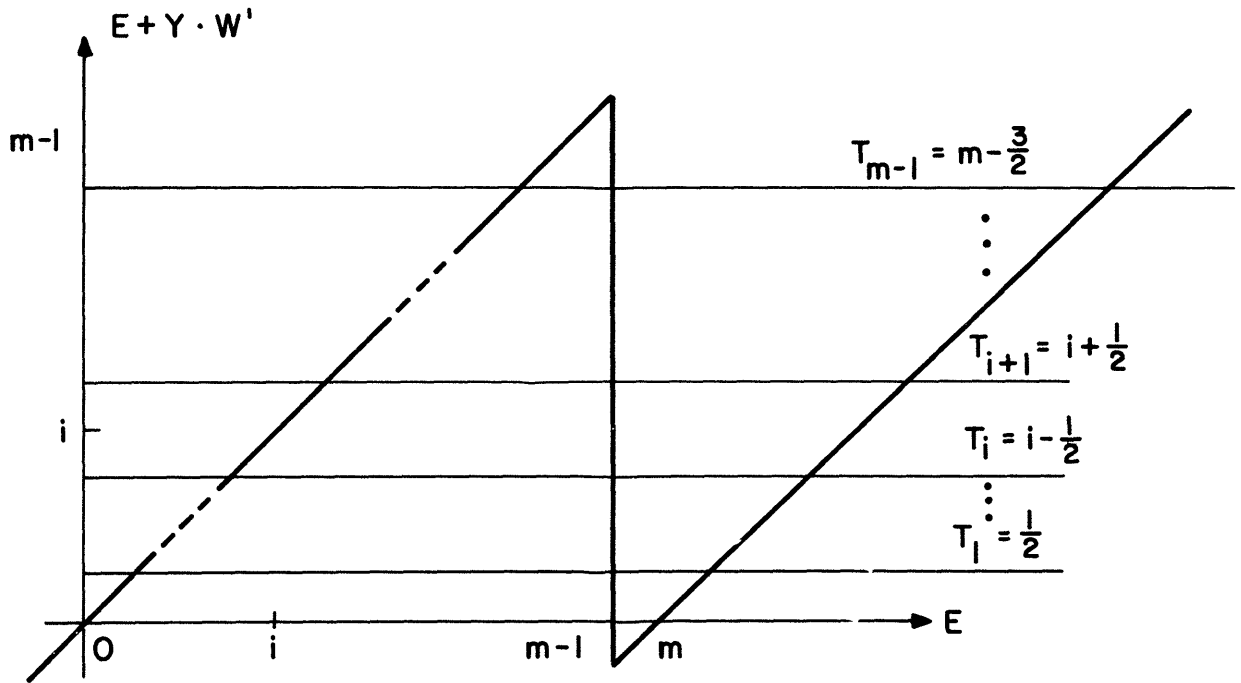


Fig. 3.3 (a) Sheng Plot for S_i

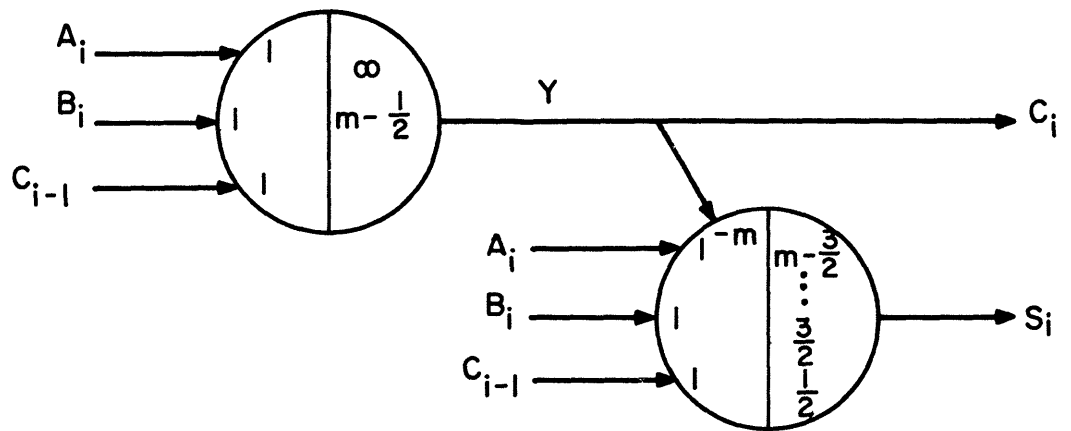


Fig. 3.3 (b) Two Level Radiation of S_i

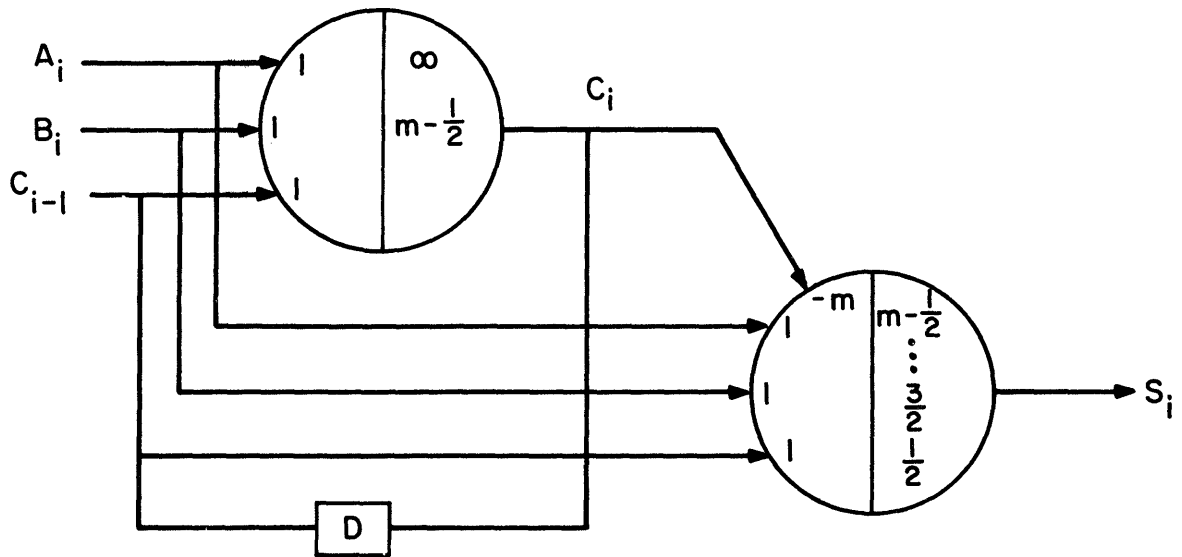


Fig. 3.4 Serial Adder for two m-ary Numbers

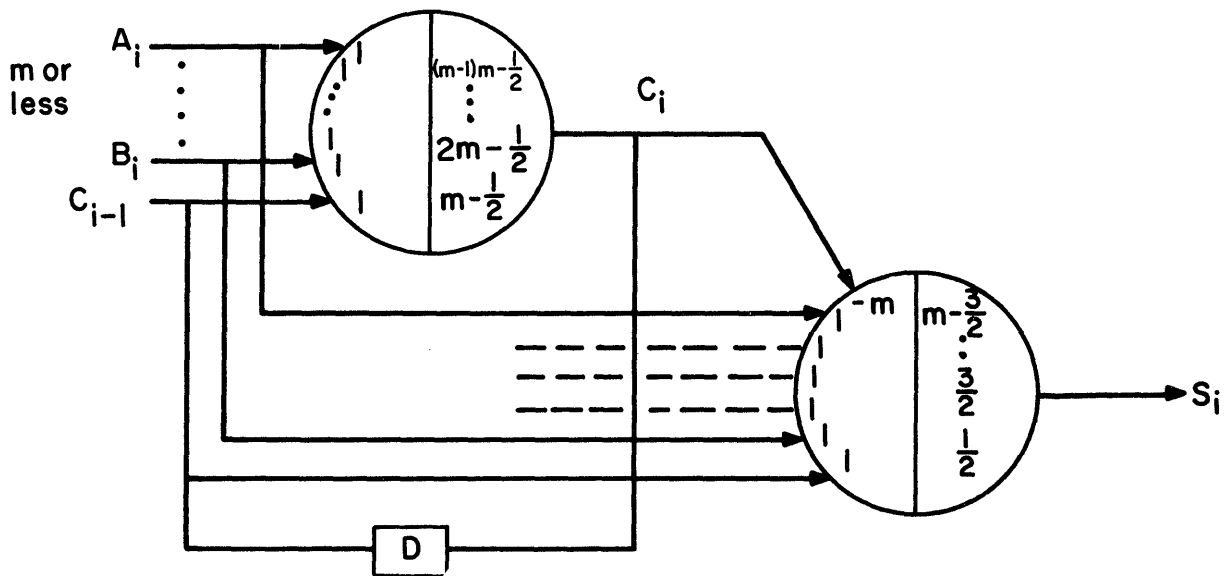


Fig. 3.5 Serial Adder for m or less m-ary Numbers

REFERENCES

1. Post, E. L, "Introduction to a General Theory of Elementary Propositions," Am. J. Math, Vol. 43, 1921.
2. Rosser, J. B. and Turquette, A. R., "Many-Valued Logics," Noth Holland, 1952.
3. di Forino, A. C., "m-valued Logics and m-ary Selection Functions," in "Automata Theory," Ed: Caianiello, E. R., Acad. Press, 1966.
4. Lowenschuss, O., "Non-Binary Switching Theory," IRE Nat. Conv. Rec., pt. 4, 1958.
5. Slupecki, J., "A Criterion of Fullness of Many-Valued Systems of Propositional Logics," Comt. rend. Soc. Lettres Varsovie, III, 32, 1939.
6. Yoeli, M. and Rosenfeld, G., "Logical Design of Ternary Switching Circuits," IEEE Trans. on Elec. Comp., Feb. 1965.
7. Lee, C. Y. and Chen, W. H., "Several Value Combinational Switching Circuits," Trans. AIEE, Vol. 75, pt. 1, 1956.
8. Berlin, R. D., "Synthesis of n-valued Switching Circuits," IRE Trans. on Elec. Comp., Vol. EC-7, 1958.
9. Gazale, M. J., "Les Structures de Commutation a m-valeur et les Calculatoires Numeriques," Paris, France: Cauthier-Villars, 1959.
10. Muhldorf, E., "Multi-Valued Switching Algebras and Their Application to Digital Systems," Proc. Nat. Elect. Conf., Vol. XV, Oct. 1959.
11. Wigington, "A New Concept in Computing," Proc. IRE, Vol. 47, 1959.
12. Menger, K. S., Jr., "Algebraic Synthesis of Modular Logic Networks," Ph.D. Thesis, Applied Math., Harvard University, June, 1966.
13. Muhldorf, E., "Ternary Switching Algebra," Archiv. der Elect. Uebertragung, Vol. 12, 1958.
14. Polansky, R. B., "Minimization of Multiple Output Switching Circuits," Comm. and Elect., AIEE, March, 1961.
15. Scheinman, A. H., "A Method for Simplifying Boolean Functions," Bell Sys. Tech. J., Vol. 41, July, 1962.
16. Diephuis, R. J., "Computer Aided Design of Threshold Element Networks," S.M. Thesis, E.E., M.I.T., Sept. 1966.

17. Sheng, C. L., "A Graphical Interpretation of Realizing Symmetric Boolean Functions with Threshold Logic Elements," Vol. EC-14, Feb. 1965.
18. Ying, C., "Studies in Multilevel Switching," S.M. Thesis, E.E., M.I.T., June 1967.

SECTION V

MODULE SYNTHESIS OF SWITCHING FUNCTIONS

S. K. Gupta and A. K. Susskind

Logic gates like AND, OR, NAND, NOR are commonly used for the synthesis of combinational networks. It is found that when these blocks are to be physically implemented as integrated chips, the size of the chip is determined by the number of pins, and so is the cost, while the circuitry is too simple to use the volume of the chip efficiently. In this section two classes of functions are analyzed for use as building blocks for combinational networks. The logical circuitry to realize both classes (modules) increases very rapidly with the number of pins. Because of the totally asymmetric nature of these modules and the complexity of the logic, these blocks are found to be difficult to use optimally, but the realizations of switching functions obtained by using these blocks require fewer blocks than when conventional gates are used. An algorithm is developed to minimize the number of pins for functions realizable by single modules. Also a non-optimum two-level synthesis procedure is introduced and a cascade of two blocks is studied for economical synthesis of four-variable functions.

CHAPTER 1

INTRODUCTION

This work is a continuation of that by Patt (Ref. 1) in which he introduced the "WOS module", which will be denoted by M_1 . He also discussed desirable properties which any module function must have (total asymmetry and logical completeness) and developed tests to check if a given function or class has these properties. Finally, he gave a procedure for using only M_1 modules in the realization of arbitrary combinational functions. Here we introduce a modified module, denoted by M_2 , and attempt to use it efficiently in the realization of arbitrary combinational functions. In the following section module M_1 is introduced and certain important results stated which are proved in Ref. 1. Finally, a brief outline of the remaining chapters is presented.

1.1 THE M_1 MODULE

Recall that every switching function can be described by a modulo-2 sum-of-products expression which is unique when the variables are used in a specified polarity (true or complemented). Such an expression is also called the irredundant ring sum expression (IRSE) and familiarity with it will be assumed. (Refs. 3, 5 and 7 may be useful in this connection.)

Definition 1.1: A well-ordered sequence (w.o.s.) is defined to be an IRSE with specified polarity of variables which has the following property. If p_i denotes a product with i variables, then all p_i contain all p_j for all j such that $2 \leq j \leq i$, where containing is defined as follows: p_a contains p_b if and only if all the variables in p_b are also in p_a .

As an immediate consequence of this definition, it can be observed that a w.o.s cannot have more than one product of a particular length except for singletons. Examples of a w.o.s. are $x_1 + x_3 + x_1x_2 + x_1x_2x_3$ and $x_1x_2x_3$, but $x_1 + x_3 + x_1x_2 + x_1x_3x_4$ and $x_1x_2 + x_1x_3$ are not well-ordered sequences.

Definition 1.2: The WOS (M_1) module (Ref. 1, page 52) of k variables, $k > 2$ is a k input, one output device which realizes the following logical expression:

$$f_k = 1 + x_1 + x_3 + x_4 + \dots + x_k + x_1x_2 + x_1x_2x_3 + \dots + x_1x_2x_3\dots x_{k-1}$$

It is proved in Ref. 1 that this module is totally asymmetric and logically complete. The following two lemmas are also proved in Ref. 1.

Lemma 1.1: Any function realized by a single M_1 module is a w.o.s.

Lemma 1.2: Every w.o.s. can be realized by a single M_1 module.

A procedure is developed in Ref. 1 for realizing an arbitrary n -variable function in two-level form using M_1 modules, where the second-level connective is EX-OR and the number of M_1 modules is minimized. The procedure simply involves splitting the IRSE of the object function into a minimum number of well-ordered sequences which when EX-ORed yield the object function.

1.2 OUTLINE OF THIS REPORT

After introducing the M_2 module in the second chapter, M_1 and M_2 are analyzed for complexity of logic. A survey of all three-variable functions is then conducted for module functions.

In the third chapter a procedure to realize all single-element realizable functions is developed which minimizes the number of pins.

In the fourth chapter all the three variable functions are shown to be realizable by a single M_2 module.

In the fifth chapter two-level and tree synthesis approaches for arbitrary n-variable functions are presented which are not optimum.

Finally, the cascade of two modules is analyzed in the sixth chapter, primarily for four-variable functions.

CHAPTER 2

THE M_2 MODULE

In this chapter a new module, denoted by M_2 , is introduced. This module is also complete and totally asymmetric. Like M_1 , module M_2 is also not defined for two variables or less, but the latter is more useful than the former, as will be shown later. The logic complexity of M_2 and M_1 as a function of the number of input pins is given in this chapter. Also, the result of an exhaustive search through all the three variable functions to find functions which could be utilized as modules is given, and their possible usefulness is briefly discussed.

2.1 INTRODUCTION TO THE M_2 MODULE

The M_2 module for n variables (denoted by M_2^n) is an n -input, one-output device defined as follows, where $+$ denotes EXCLUSIVE-OR:

$$\begin{aligned}
 M_2^n = 1 + x_2 + & \left\{ \begin{array}{l} x_1x_3 + x_1x_4 + x_1x_5 + \dots + x_{n-1}x_n \\ \text{(all the products of length two except } x_1x_2) \end{array} \right. & (2.1) \\
 & + \left\{ \begin{array}{l} x_1x_3x_4 + x_1x_3x_5 + x_1x_3x_6 + \dots + x_{n-2}x_{n-1}x_n \\ \text{(all the products of length three except } x_1x_2x_3) \end{array} \right. \\
 & + \dots \\
 & \dots \\
 & \dots \\
 & + \left\{ \begin{array}{l} x_1x_3x_4 \dots x_n + x_1x_2x_4 \dots x_n + \dots + x_2x_3x_4 \dots x_n \\ \text{(all the products of length } (n-1) \text{ except } x_1x_2x_3 \dots x_{n-1}) \end{array} \right.
 \end{aligned}$$

The above is the unique irredundant modulo-2 sum-of-products expression, where x_i denotes the variable connected to the i^{th} pin of the module.

For later convenience, we rewrite (2.1) as follows. Let A^n denote the n-variable Boolean **product** $\bar{x}_1\bar{x}_2\cdots\bar{x}_n$. Then A^n can be written in ring-sum form as

$$\begin{aligned} A^n &= 1 + x_1 + x_2 + x_3 + \dots \\ &\quad + x_1x_2 + x_1x_3 + x_1x_4 + \dots \\ &\quad + x_1x_2x_3 + x_1x_2x_4 + \dots \\ &\quad \dots \\ &\quad \dots \\ &\quad + x_1x_2x_3\cdots x_n \end{aligned}$$

and $M_2^n + A^n = x_1 + x_3 + x_4 + \dots + x_n + x_1x_2 + x_1x_2x_3 + \dots +$
 $+ x_1x_2x_3\cdots x_{n-1} + x_1x_2x_3\cdots x_n$

Now by definition of M_1^n (see Chapter 1) we have

$$M_2^n + A^n = 1 + M_1^n + x_1x_2x_3\cdots x_n$$

or $M_2^n = 1 + M_1^n + \bar{x}_1\bar{x}_2\bar{x}_3\cdots\bar{x}_n + x_1x_2x_3\cdots x_n$ (2.2)

This form is very handy and will be used frequently.

2.2 COMPLETENESS AND ASYMMETRY OF M_2^n

Theorem 2.1: M_2^n is a logically complete function without biasing.

To prove this theorem recall the three necessary and sufficient conditions for completeness without biasing (Ref. 1):

- i) The vertex $\bar{x}_1\bar{x}_2\bar{x}_3\cdots\bar{x}_n$ (decimal code 0) must be true.
- ii) The vertex $x_1x_2x_3\cdots x_n$ (decimal code 2^n-1) must be false.
- iii) At least two complementary vertices in the truth table of the function must have identical outputs.

Proof:

- i) The presence of 1 in the definition of M_2^n (2.1) makes the vertex $\bar{x}_1\bar{x}_2\bar{x}_3\cdots\bar{x}_n$ true, since all product terms are 0 when the

substitution $x_i = 0$ ($i = 1$ to n) is made.

ii) To prove that the vertex $x_1x_2x_3\cdots x_n$ is always false, we proceed as follows.

$$M_2^n = 1 + M_1^n + x_1x_2x_3\cdots x_n + \bar{x}_1\bar{x}_2\bar{x}_3\cdots\bar{x}_n$$

$$\text{Hence } M_2^n (2^n - 1) = 1 + M_1^n (2^n - 1) + 1 + 0.$$

As M_1^n is logically complete, $M_1^n(2^n - 1) = 0$. Therefore, $M_2^n(2^n - 1) = 0$.

iii) Because M_1^n is logically complete, there exist two complementary vertices P and Q such that $M_1^n(P) = M_1^n(Q)$. Also $P \neq \bar{x}_1\bar{x}_2\bar{x}_3\cdots\bar{x}_n$, because $M_1^n(0) \neq M_1^n(2^n - 1)$. Therefore, in P , for some x_i and x_j , $x_i = 1$ and $x_j = 0$ such that in Q , $x_j = 1$ and $x_i = 0$. Then $M_2^n(P) = 1 + M_1^n(P) + 0 + 0 = 1 + M_1^n(P)$, and similarly $M_2^n(Q) = 1 + M_1^n(Q)$. But because $M_1^n(P) = M_1^n(Q)$, $M_2^n(P) = M_2^n(Q)$, so that we can conclude that M_2^n is complete without biasing.

Theorem 2.2: M_2^n is a totally asymmetric function for all values of n .

Proof: This proof is based on a result obtained by Patt (Ref. 1). We proceed with the construction of the β vectors defined as follows: β_{ij} (the j^{th} element in the i^{th} β -vector) is defined to be the number of terms of length j which contain the variable x_i in the irredundant ring sum expression (IRSE) of the function.

Now from the definition (2.1) of M_2^n we have

$$\begin{aligned}
 \beta_1 &= 0, \binom{n-1}{1} - 1, \binom{n-1}{2} - 1, \dots \dots \binom{n-1}{j-1} - 1, \dots \binom{n-1}{n-2} - 1, 0 \\
 \beta_2 &= 1, \binom{n-1}{1} - 1, \binom{n-1}{2} - 1, \dots \dots \binom{n-1}{j-1} - 1, \dots \binom{n-1}{n-2} - 1, 0 \\
 \beta_3 &= 0, \binom{n-1}{1} \quad , \quad \binom{n-1}{2} - 1, \dots \dots \binom{n-1}{j-1} - 1, \dots \binom{n-1}{n-2} - 1, 0 \\
 \beta_4 &= 0, \binom{n-1}{1} \quad , \quad \binom{n-1}{2} \quad , \dots \dots \binom{n-1}{j-1} - 1, \dots \binom{n-1}{n-2} - 1, 0 \\
 &\dots \\
 &\dots \\
 &\dots \\
 \beta_j &= 0, \binom{n-1}{1} \quad , \quad \binom{n-1}{2} \quad , \dots \dots \binom{n-1}{j-1} - 1, \dots \binom{n-1}{n-2} - 1, 0 \\
 &\dots \\
 &\dots \\
 &\dots \\
 \beta_n &= 0, \binom{n-1}{1} \quad , \quad \binom{n-1}{2} \quad , \dots \dots \binom{n-1}{j-1} \quad , \dots \binom{n-1}{n-2} \quad , 0
 \end{aligned}$$

All these vectors are different from each other which, as shown by Patt, is sufficient to prove that M_2^n is totally asymmetric.

2.3 LOGIC COMPLEXITY OF M_2^n AND M_1^n

By logic complexity of a module we mean the quantity of circuitry required to realize that module physically. A rough measure of this is the number of AND and OR gates in a 2-level minimum realization. To study this, we build up a recursive construction of these modules.

Recursive Construction of M_1^n

Here and henceforth, we use + to denote EXCLUSIVE-OR and U to denote INCLUSIVE-OR.

$$\begin{aligned}
 M_1^n &= 1 + x_1 + x_2 + x_3 + \dots + x_n + x_1x_2 + x_1x_2x_3 + \dots + x_1x_2x_3 \dots x_{n-1} \\
 &= M_1^{n-1} + x_1x_2x_3 \dots x_{n-1} + x_n \\
 &= \overline{x}_n (M_1^{n-1} + x_1x_2x_3 \dots x_{n-1}) + x_n (1 + M_1^{n-1} + x_1x_2x_3 \dots x_{n-1}) \\
 &= \overline{x}_n (M_1^{n-1} + x_1x_2x_3 \dots x_{n-1}) \cup x_n (1 + M_1^{n-1} + x_1x_2x_3 \dots x_{n-1})
 \end{aligned}$$

Let $P^n = M_1^{n-1} + x_1x_2x_3 \dots x_{n-1}$ so that function P^n is just M_1^{n-1} with its value at vertex $x_1x_2x_3 \dots x_{n-1}$ complemented, i.e., made true.

Then

$$M_1^n = \overline{x}_n P^n \cup x_n \overline{P^n} \quad (2.3)$$

Let $T(G)$ represent the set of true vertices of function G and $F(G)$ represent the set of false vertices of function G . The elements of these sets will be designated by the usual decimal notation, i.e., the additive weight of variable x_i is 2^{i-1} , $1 \leq i \leq n$. Also, let $W[T(G)]$ represent the set of true vertices of function G incremented by decimal weight W . Using U to denote set union, it follows from (2.3) that

$$T(M_1^n) = T(M_1^{n-1}) \cup \{2^{n-1}-1\} \cup 2^{n-1}[F(M_1^{n-1})] - \{2^n-1\} \quad (2.4)$$

Relation (2.4) states that the true body of M_1^n consists of (a) the true body of M_1^{n-1} , (b) the vertex $2^{n-1}-1$, and (c) the entire false body of M_1^{n-1} , relabelled by the additive constant 2^{n-1} except for vertex 2^n-1 . For example, for module M_1^3 , $T(M_1^3) = \{0,2,3,5\}$ and $F(M_1^3) = \{1,4,6,7\}$. Hence

$$\begin{aligned}
 T(M_1^4) &= \{0,2,3,5\} \cup \{7\} \cup 8 [1,4,6,7] - \{15\} \\
 &= \{0,2,3,5,7,9,12,14\}
 \end{aligned}$$

Since M_1^3 has four true (and four false) vertices, it is clear from this

observation that

Lemma 2.1: The true body of M_1^n consists of 2^{n-1} vertices, for all n .

In other words, M_1^n is a "50-50" function.

In terms of Karnaugh maps, we can interpret (2.4) as follows. Function M_1^n can be represented by two maps: (1) The maps of the true body of M_1^{n-1} with vertex $2^{n-1}-1$ made true. (This corresponds to the first two terms on the right and gives the true body of M_1^n with $x_n = 0$.) And (2) the map of the false body of M_1^{n-1} with vertex $2^{n-1}-1$ omitted (made false) and all vertex labels incremented by 2^{n-1} . (This corresponds to the last two terms on the right and gives the true body of M_1^n with $x_n = 1$.)

Lemma 2.2: Complementing the vertex $x_1x_2x_3\dots x_n$ in M_1^n and in $(1 + M_1^n + x_1x_2x_3\dots x_n)$ does not give rise to any prime implicants covering more than two vertices.

Proof: (By induction) Basis: The lemma is true for $n=3$ (see Fig. 2.1). Assume that the lemma is true for $n-1$.

a) Complementing vertex 2^{n-1} in M_1^n gives $M_1^n + x_1x_2x_3\dots x_n$.

$$(M_1^n + x_1x_2x_3\dots x_n) \Big|_{x_n=0} = M_1^n \Big|_{x_n=0} = M_1^{n-1} + x_1x_2x_3\dots x_{n-1}$$

$$(M_1^n + x_1x_2x_3\dots x_n) \Big|_{x_n=1} = 1 + M_1^{n-1} + x_1x_2x_3\dots x_{n-1} + x_1x_2x_3\dots x_{n-1}$$

By induction assumption, in $M_1^{n-1} + x_1x_2x_3\dots x_{n-1}$ no implicant of more than two vertices is generated by the addition of $x_1x_2x_3\dots x_{n-1}$ to M_1^{n-1} . Similarly, no implicant of more than two vertices is generated in $(M_1^n + x_1x_2x_3\dots x_n) \Big|_{x_n=1}$. Now in the map of $M_1^n + x_1x_2x_3\dots x_n$ the

vertex $x_1x_2x_3\dots x_n$ can generate an implicant of more than two ones if

x_1	$x_2 x_3$	00	01	11	10
0		1	1		
1			1		1

Fig. 2.1a M_1^3

x_1	$x_2 x_3$	00	01	11	10
0		1	1		
1			1	1	1

Fig. 2.1b $M_1^3 + x_1 x_2 x_3$

x_1	$x_2 x_3$	00	01	11	10
0				1	1
1		1			

Fig. 2.1c $1 + M_1^3 + x_1 x_2 x_3$

x_1	$x_2 x_3$	00	01	11	10
0				1	1
1		1		1	

Fig. 2.1d $1 + M_1^3$

Fig. 2.1 Maps of M_1^3 and Related Functions

and only if some true vertex adjacent to $x_1x_2x_3\cdots x_{n-1}$ in

$$(M_1^n + x_1x_2x_3\cdots x_n) \Big|_{x_n=1} \text{ has its image true in } (M_1^n + x_1x_2x_3\cdots x_n) \Big|_{x_n=0}.$$

But since the two maps are mutual complements except at $x_1x_2x_3\cdots x_{n-1}$, this is not possible and therefore complementing $x_1x_2x_3\cdots x_n$ in M_1^n cannot generate an implicant of more than two ones.

b) Complementing the vertex $x_1x_2x_3\cdots x_n$ in $1 + M_1^n + x_1x_2x_3\cdots x_n$ gives $1 + M_1^n$ and

$$(1 + M_1^n) \Big|_{x_n=0} = (1 + M_1^n + x_1x_2x_3\cdots x_n) \Big|_{x_n=0} = 1 + M_1^{n-1} + x_1x_2x_3\cdots x_{n-1}$$

$$(1 + M_1^n) \Big|_{x_n=1} = M_1^{n-1} + x_1x_2x_3\cdots x_{n-1}$$

By induction assumption, vertex $x_1x_2x_3\cdots x_{n-1}$ when complemented in M_1^{n-1} does not give rise to an implicant of more than two ones in

$$(1 + M_1^n) \Big|_{x_n=1}. \text{ The vertex } x_1x_2x_3\cdots x_{n-1} \text{ in } (1 + M_1^n) \Big|_{x_n=0} \text{ is false.}$$

Therefore in the whole map of $(1 + M_1^n)$ complementing vertex 2^n-1 does not give rise to an implicant of more than two ones.

Lemma 2.3: M_1^n and $(1 + M_1^n + x_1x_2x_3\cdots x_n)$ have no prime implicant covering more than two vertices.

Proof: (By induction) Basis: For M_1^3 both parts are true (Fig. 2.1a and 2.1c). Assume that the lemma is true for M_1^{n-1} .

As in the proof of Lemma 2.2, $M_1^n \Big|_{x_n=0} = M_1^{n-1} + x_1x_2\cdots x_{n-1}$ and, by the assumption and Lemma 2.2, it does not have any prime implicant that covers more than two vertices. Also $M_1^n \Big|_{x_n=1} = 1 + M_1^{n-1} + x_1x_2\cdots x_{n-1}$. By assumption, this does not have any prime implicant that covers more than two vertices. But $M_1^n \Big|_{x_n=0}$ and $M_1^n \Big|_{x_n=1}$ are

mutually complementary, and hence cannot jointly generate larger prime implicants. This proves the first part of the lemma.

$$\text{Similarly, } (1 + M_1^n + x_1 x_2 \dots x_n) \Big|_{x_n=0} = 1 + M_1^{n-1} + x_1 x_2 \dots x_{n-1},$$

which by induction assumption has no prime implicants covering more than two vertices. Also $(1 + M_1^n + x_1 x_2 \dots x_n) \Big|_{x_n=1} = M_1^{n-1}$, which is also assumed not to have any prime implicants covering more than two vertices. Now M_1^{n-1} and $1 + M_1^{n-1} + x_1 x_2 \dots x_{n-1}$ are mutually complementary except at vertex $x_1 x_2 \dots x_{n-1}$, where both are false. Hence $1 + M_1^n + x_1 x_2 \dots x_n$ has no prime implicants covering more than two vertices.

Let $N(F)$ denote the number of AND gates required to realize function F in two-level form.

Lemma 2.4: $N(M_1^n + x_1 x_2 x_3 \dots x_n) = N(M_1^n)$ for odd n .

Proof: By definition,

$$M_1^n = 1 + x_1 + x_3 + x_4 + \dots + x_n + x_1 x_2 + x_1 x_2 x_3 + x_1 x_2 x_3 \dots x_{n-1}$$

The vertex $x_1 \bar{x}_2 x_3 x_4 \dots x_n$ is one of the vertices which is adjacent to the vertex $x_1 x_2 x_3 \dots x_n$, and the former is a true vertex in M_1^n for odd n since all products vanish and the number of singletons is even. Any vertex adjacent to $x_1 \bar{x}_2 x_3 x_4 \dots x_n$ other than $x_1 x_2 x_3 \dots x_n$ can be written in general as $x_1 \bar{x}_2 x_3 \dots \bar{x}_i \dots x_n$ and by direct substitution is seen to be false for all values of i . Thus the vertex $x_1 \bar{x}_2 x_3 \dots x_n$ is an isolated true vertex adjacent to $x_1 x_2 x_3 \dots x_n$, and when the vertex $x_1 x_2 x_3 \dots x_n$ is complemented to make it true, it merges with $x_1 \bar{x}_2 x_3 \dots x_n$ without costing any extra gate, in fact saving a pin.

Lemma 2.5: $N(M_1^n + x_1x_2x_3\dots x_n) = N(M_1^n) + 1$ for even n .

Proof:

$$M_1^n = 1 + x_1 + x_3 + x_5 + \dots + x_{n-1} + x_n + x_1x_2 + x_1x_2x_3 + \dots + x_1x_2x_3\dots x_{n-1}$$

Any vertex with a single variable complemented is a vertex adjacent to $x_1x_2x_3\dots x_n$. Out of all these vertices the following are true:

$$\begin{array}{l} \bar{x}_1x_2x_3 \dots x_n \\ x_1x_2x_3\bar{x}_4 \dots x_n \\ x_1x_2x_3x_4x_5\bar{x}_6 \dots x_n \\ \dots \\ \dots \\ x_1x_2x_3 \dots \bar{x}_n \end{array}$$

It is easy to check by direct substitution that all other vertices unit-distant from $x_1x_2\dots x_n$ are false. For each of the above true vertices $x_1x_2x_3\dots\bar{x}_i\dots x_n$, consider the vertex $x_1\bar{x}_2x_3\dots\bar{x}_i\dots x_n$. This vertex is also true, since n is even and there is an even number of singletons left when this vertex is evaluated. This vertex for every even i is such that any other vertex adjacent to it is false, since by substituting $x_1\bar{x}_2x_3\dots\bar{x}_j\dots\bar{x}_i\dots x_n$ another singleton is made zero, changing the value of the function. Therefore all these vertices are such that they have only one vertex adjacent to them true. That vertex is one from the list of true vertices adjacent to $x_1x_2x_3\dots x_n$. This implies that no matter which of these vertices is chosen to merge with $x_1x_2x_3\dots x_n$, it will cost an extra gate.

Lemma 2.6: $N(1 + M_1^n + x_1x_2\dots x_n) = N(M_1^n) - 1$ for n odd.

and $N(1 + M_1^n + x_1x_2\dots x_n) = N(M_1^n)$ for n even.

Proof: (By induction) Basis: The lemma is true for $n = 3$ and $n = 4$, as can be seen from Figs. 2.2. Assume the lemma is true for some even n and corresponding $n-1$.

The function M_1^{n+1} can be decomposed into two parts as follows:

$$M_1^{n+1} \Big|_{x_{n+1}=0} = M_1^n + x_1 x_2 x_3 \dots x_n$$

$$\text{and } M_1^{n+1} \Big|_{x_{n+1}=1} = 1 + M_1^n + x_1 x_2 x_3 \dots x_n$$

Since these two parts are mutually complementary,

$$N(M_1^{n+1}) = N(M_1^{n+1} \Big|_{x_{n+1}=0}) + N(M_1^{n+1} \Big|_{x_{n+1}=1})$$

(Here and elsewhere the + sign is also used to denote the ordinary addition of numbers.)

By Lemma 2.5

$$N(M_1^n + x_1 x_2 x_3 \dots x_n) = N(M_1^n) + 1 \quad (\text{for even } n)$$

and by induction assumption

$$N(1 + M_1^n + x_1 x_2 x_3 \dots x_n) = N(M_1^n)$$

$$\text{Therefore} \quad N(M_1^{n+1}) = 2 N(M_1^n) + 1 \quad (2.5)$$

Now let

$$F = 1 + M_1^{n+1} + x_1 x_2 x_3 \dots x_{n+1}$$

$$F \Big|_{x_{n+1}=0} = 1 + M_1^n + x_1 x_2 x_3 \dots x_n$$

$$F \Big|_{x_{n+1}=1} = M_1^n$$

These two parts of F are mutually complementary except at the vertex $x_1 x_2 x_3 \dots x_n$, where both are false. Therefore, the following is still true

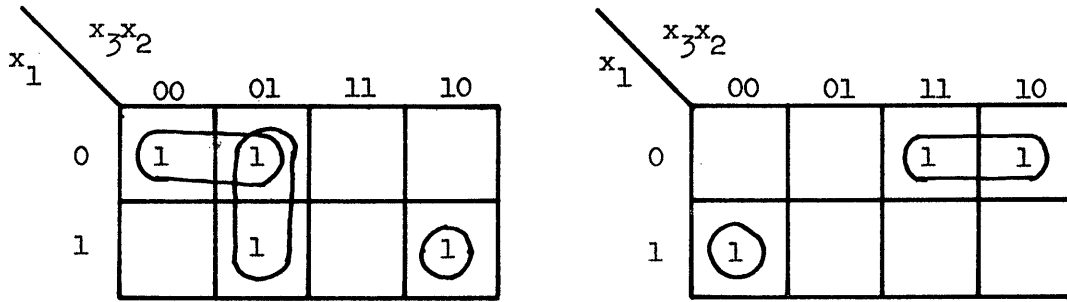


Fig. 2.2a $N(M_1^3) = 3$ and $N(1 + M_1^3 + x_1x_2x_3) = 2$

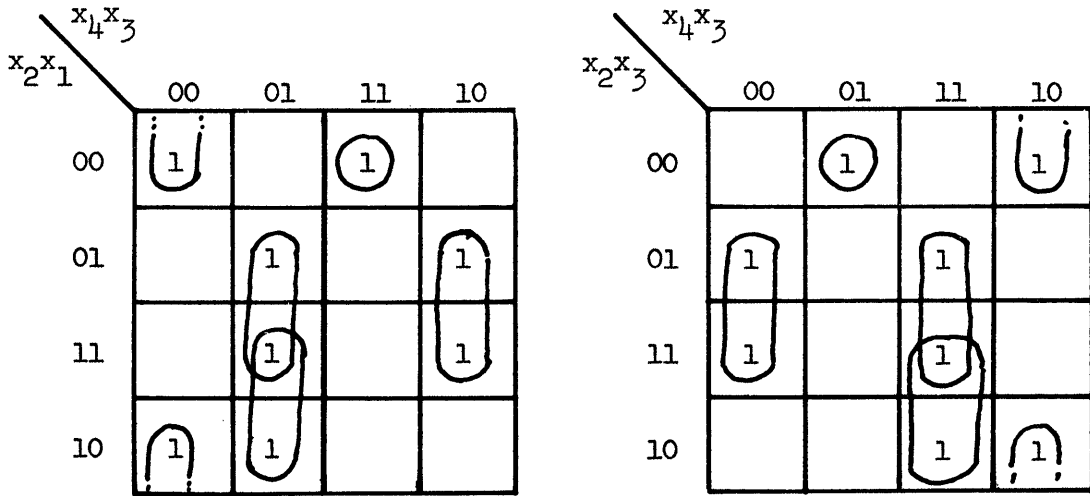


Fig. 2.2b $N(M_1^4) = 5$ and $N(1 + M_1^4 + x_1x_2x_3x_4) = 5$

Fig. 2.2 Maps of M_1^3 , M_1^4 , and Related Function

$$\begin{aligned} N(F) &= N(F|_{x_{n+1}=0}) + N(F|_{x_{n+1}=1}) \\ &= N(1 + M_1^n + x_1 x_2 x_3 \dots x_n) + N(M_1^n) \end{aligned}$$

So by induction assumption

$$N(F) = 2N(M_1^n) \tag{2.6}$$

and by (2.5) and (2.6)

$$N(1 + M_1^{n+1} + x_1 x_2 x_3 \dots x_{n+1}) = N(M_1^{n+1}) - 1$$

This proves the first claim.

Now $(n+2)$ is even and M_1^{n+2} can be divided into two parts as follows.

$$\begin{aligned} M_1^{n+2} |_{x_{n+2}=0} &= M_1^{n+1} + x_1 x_2 x_3 \dots x_{n+1} \\ M_1^{n+2} |_{x_{n+2}=1} &= 1 + M_1^{n+1} + x_1 x_2 x_3 \dots x_{n+1} \end{aligned}$$

Since the two are mutual complements

$$\begin{aligned} N(M_1^{n+2}) &= N(M_1^{n+2} |_{x_{n+2}=0}) + N(M_1^{n+2} |_{x_{n+2}=1}) \\ &= N(M_1^{n+1} + x_1 x_2 x_3 \dots x_{n+1}) + N(1 + M_1^{n+1} + x_1 x_2 x_3 \dots x_{n+1}) \end{aligned}$$

By Lemma 2.4 ($n+1$ being odd) and the first part of the proof

$$\begin{aligned} N(M_1^{n+2}) &= N(M_1^{n+1}) + N(M_1^{n+1}) - 1 \\ &= 2N(M_1^{n+1}) - 1 \end{aligned} \tag{2.7}$$

Also $1 + M_1^{n+2} + x_1 x_2 x_3 \dots x_{n+2}$ can be divided into two parts.

$$\text{Let } F_1 = 1 + M_1^{n+2} + x_1 x_2 x_3 \dots x_{n+2}$$

$$\begin{aligned} F_1 |_{x_{n+2}=0} &= 1 + M_1^{n+1} + x_1 x_2 x_3 \dots x_{n+1} \\ F_1 |_{x_{n+2}=1} &= M_1^{n+1} \end{aligned}$$

By reasoning similar to that of before

$$\begin{aligned}
 N(F_1) &= N(F_1 |_{x_{n+2}=0}) + N(F_1 |_{x_{n+2}=1}) \\
 &= N(M_1^{n+1}) + N(1 + M_1^{n+1} + x_1 x_2 x_3 \dots x_{n+1}) \\
 &= 2N(M_1^{n+1}) - 1 \tag{2.8}
 \end{aligned}$$

And from (2.7) and (2.8), for even $(n+2)$,

$$N(1 + M_1^{n+2} + x_1 x_2 \dots x_{n+2}) = N(M_1^{n+2})$$

This proves the second claim. The following theorem then follows:

Theorem 2.3: The number of 'AND' gates required in 2-level minimal 'AND-OR' synthesis of M_1^n , $N(M_1^n)$, is

1. n odd, $N(M_1^n) = 2^{n-2} + 2^{n-4} + \dots + 2^1 + 1$
2. n even, $N(M_1^n) = 2^{n-2} + 2^{n-4} + \dots + 2^2 + 1$

Proof: (By induction) Basis: From Fig. 2.2, for $n = 3$ and 4 ,

$$\begin{aligned}
 N(M_1^3) &= 3 = 2^1 + 1 \\
 N(M_1^4) &= 5 = 2^2 + 1
 \end{aligned}$$

Assume the theorem is true for some n .

$$\begin{aligned}
 \text{Since } M_1^{n+1} |_{x_{n+1}=0} &= 1 + M_1^{n+1} |_{x_{n+1}=1} \\
 N(M_1^{n+1}) &= N(M_1^{n+1} |_{x_{n+1}=0}) + N(M_1^{n+1} |_{x_{n+1}=1}) \\
 &= N(M_1^n + x_1 x_2 x_3 \dots x_n) + N(1 + M_1^n + x_1 x_2 x_3 \dots x_n)
 \end{aligned}$$

For n even, by Lemmas 2.5 and 2.6

$$\begin{aligned}
 N(M_1^{n+1}) &= N(M_1^n) + 1 + N(M_1^n) \\
 &= 2N(M_1^n) + 1 \\
 &= 2(2^{n-2} + 2^{n-4} + \dots + 2^2 + 1) + 1 \\
 &= 2^{(n+1)-2} + 2^{(n+1)-4} + \dots + 2^1 + 1
 \end{aligned}$$

and for n odd, by Lemmas 2.4 and 2.6,

$$\begin{aligned} N(M_1^{n+1}) &= N(M_1^n) + N(M_1^n) - 1 \\ &= 2N(M_1^n) - 1 \\ &= 2(2^{n-2} + 2^{n-4} + \dots + 2^{1+1}) - 1 \\ &= 2^{(n+1)-2} + 2^{(n+1)-4} + \dots + 2^{2+1} \end{aligned}$$

The following two conclusions can be reached on the basis of Theorem 2.3 and Lemma 2.3.

1. In a two level 'AND-OR' minimal realization the AND gates have at least (n-1) pins.

2. The number of AND gates is almost doubled with the addition of a single input pin.

Similar results can be obtained about M_2^n .

Recursive Construction of M_2^n

$$\begin{aligned} \text{From (2.2), } M_2^n &= 1 + M_1^n + x_1 x_2 x_3 \dots x_n + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n \\ M_2^n &= 1 + M_1^{n-1} + x_n \cdot x_1 x_2 x_3 \dots x_{n-1} + x_1 x_2 x_3 \dots x_n + \\ &\quad + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n \\ &= 1 + M_1^{n-1} + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_{n-1} + x_1 x_2 x_3 \dots x_{n-1} + \\ &\quad + x_n (1 + x_1 x_2 x_3 \dots x_{n-1} + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_{n-1}) \\ &= M_2^{n-1} + x_n (1 + x_1 x_2 x_3 \dots x_{n-1} + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_{n-1}) \\ &= \bar{x}_n M_2^{n-1} + x_n (1 + M_2^{n-1} + x_1 x_2 \dots x_{n-1} + \bar{x}_1 \bar{x}_2 \dots \bar{x}_{n-1}) \\ &= \bar{x}_n M_2^{n-1} \cup x_n (1 + M_2^{n-1} + x_1 x_2 \dots x_{n-1} + \bar{x}_1 \bar{x}_2 \dots \bar{x}_{n-1}) \end{aligned}$$

Using the vertex notation introduced before

$$T(M_2^n) = T(M_2^{n-1}) \cup 2^{n-1} [F(M_2^{n-1})] - \{2^{n-1}\} \cup \{2^{n-1}\}$$

For example,

$$M_2^3 = 1 + x_2 + x_1 x_3 + x_2 x_3$$

With the weight assignment $w_i = 2^{i-1}$,

$$T(M_2^3) = \{0, 1, 4, 6\} \quad \text{and} \quad F(M_2^3) = \{2, 3, 5, 7\}$$

$$\begin{aligned} T(M_2^4) &= \{0, 1, 4, 6\} \cup \{10, 11, 13, 15\} - \{15\} \cup \{8\} \\ &= \{0, 1, 4, 6, 8, 10, 11, 13\} \end{aligned}$$

As for M_1^n , the following lemmas are easy to prove:

Lemma 2.7: The true body of M_2^n consists of 2^{n-1} vertices, for all n .

Lemma 2.8: There is no prime implicant with more than two ones in M_2^n .

Also, the number of gates required to physically realize M_2^n grows as it does in M_1^n : it almost doubles with every input pin added.

Looking at how the complexity of logic grows with the number of pins, one doubts that the number of pins would be the limiting criterion if the circuit is realized as an integrated chip, as it usually is. Because every added pin dictates that the logic in the circuit be increased by a large amount, the logic density may be the determining factor.

2.4 SEARCH OF ALL THREE-VARIABLE FUNCTIONS FOR MODULE FUNCTIONS

As mentioned in Chapter 1, Patt (Ref. 1) has pointed out that total asymmetry and completeness are very desirable properties in a class of functions to be used as modules. What follows is a list of all equivalence classes of three-variable functions and the result of

an exhaustive examination for functions having these two properties.

Functions of two variables and less are omitted.

No.	Representative Function	Totally Asymmetric?	Complete?
1.	$1 + x_1x_2x_3$	NO (123)	
2.	$1 + x_1 + x_1x_2x_3$	NO (23)	
3.	$1 + x_1 + x_2 + x_1x_2x_3$	NO (12)	
4.	$1 + x_1 + x_2 + x_3 + x_1x_2x_3$	NO (123)	
5.	$1 + x_1 + x_2 + x_3$	NO (123)	
6.	$1 + x_1x_2 + x_1x_2x_3$	NO (23)	
7.	$1 + x_1x_2 + x_2x_3 + x_1x_2x_3$	NO (123)	
8.	$1 + x_1x_2 + x_2x_3 + x_3x_1 + x_1x_2x_3$	NO (123)	
9.	$1 + x_1x_2 + x_2x_3 + x_1x_3$	NO (123)	
10.	$1 + x_1x_2 + x_1x_3$	NO (23)	
11.	$1 + x_1 + x_2x_3$	NO (23)	
12.	$1 + x_1 + x_1x_2 + x_2x_3$	YES	YES M_2^3
13.	$1 + x_1 + x_1x_2 + x_1x_3$	NO (23)	
14.	$1 + x_1 + x_1x_2 + x_3x_2 + x_1x_3$	NO (23)	
15.	$1 + x_1 + x_2 + x_2x_3$	YES	YES M_1^3
16.	$1 + x_1 + x_2 + x_1x_2 + x_2x_3$	YES	NO
17.	$1 + x_1 + x_2 + x_1x_3 + x_2x_3$	NO (12)	
18.	$1 + x_1 + x_2 + x_1x_2 + x_2x_3 + x_1x_3$	NO (12)	
19.	$1 + x_1 + x_2 + x_3 + x_1x_2$	NO (12)	
20.	$1 + x_1 + x_2 + x_3 + x_1x_2 + x_1x_3$	NO (23)	
21.	$1 + x_1 + x_2 + x_3 + x_1x_2 + x_1x_3 + x_2x_3$	NO (123)	
22.	$1 + x_1 + x_2x_3 + x_1x_2x_3$	NO (23)	
23.	$1 + x_1 + x_1x_2 + x_1x_2x_3$	YES	YES

No.	Representative Function	Totally Asymmetric?	Complete?
24.	$1 + x_1 + x_1x_2 + x_1x_3 + x_1x_2x_3$	NO (23)	
25.	$1 + x_1 + x_1x_2 + x_2x_3 + x_1x_2x_3$	YES	NO
26.	$1 + x_1 + x_1x_2 + x_2x_3 + x_2x_1 + x_1x_2x_3$	NO (23)	
27.	$1 + x_1 + x_2 + x_2x_3 + x_1x_2x_3$	YES	NO
28.	$1 + x_1 + x_2 + x_1x_2 + x_1x_2x_3$	NO (12)	
29.	$1 + x_1 + x_2 + x_1x_2 + x_2x_3 + x_1x_2x_3$	YES	YES
30.	$1 + x_1 + x_2 + x_1x_3 + x_2x_3 + x_1x_2x_3$	NO (12)	
31.	$1 + x_1 + x_2 + x_1x_2 + x_1x_3 + x_2x_3 + x_1x_2x_3$	NO (12)	
32.	$1 + x_1 + x_2 + x_3 + x_2x_3 + x_1x_2x_3$	NO (23)	
33.	$1 + x_1 + x_2 + x_3 + x_1x_2 + x_2x_3 + x_1x_2x_3$	NO (13)	
34.	$1 + x_1 + x_2 + x_3 + x_1x_2 + x_2x_3 + x_2x_1 + x_1x_2x_3$	NO (123)	

Numbers 15 and 12 are just M_1^3 and M_2^3 , respectively. Numbers 23 and 29 represent the only other two classes which meet the requirements of completeness and total asymmetry. But these are only three-variable functions, and it is difficult to suggest how these would lead to a recursive definition for all n . One possible recursion on each of these classes is as follows.

Let F_1^n represent the class obtained from No. 23 and F_2^n represent the class obtained from No. 29.

$$F_1^n = M_1^n + x_n + x_1x_2x_3 \dots x_n$$

$$F_2^n = M_2^n + x_1 + x_1x_2x_3 \dots x_n$$

It is easy to see that these recursions preserve the properties of completeness and total asymmetry, but are trivially close to M_1 and

M_2 , which means that prospective usefulness of F_1 is almost the same as that of M_1 , and so is that of F_2 when compared with M_2 .

While we have given some consideration to functions of four variables, the profusion of candidates has prevented us from achieving meaningful results.

CHAPTER 3
OPERATIONS ON M_1^n AND M_2^n

By biasing module input pins to Boolean constants and by connecting a variable to more than one pin, a module can realize several classes of functions. Here we will analyze the effects of these operations. Also, an algorithm for synthesizing single-element realizable functions by M_2^n is presented which minimizes the number of input pins. Several functions like 'AND', 'OR', and 'Ex-OR' are then realized by a single module. Finally, it is shown how modules with $(n+1)$ pins could be realized using modules with n pins or less.

3.1 BIASING AND DUPLICATING OPERATIONS

The effects of biasing and duplicating pins on M_1^n are not explicitly discussed in Ref. 1, but their extensive use makes these effects easily understandable. We shall discuss the effect of these operations on M_2^n , and the corresponding effects on M_1^n follow from these.

Biasing the i^{th} Pin of M_2^n to 0

Recall that

$$\begin{aligned} M_2^n &= 1 + M_1^n + x_1 x_2 x_3 \dots x_n + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n \\ &= x_1 + x_3 + x_4 + \dots + x_n + x_1 x_2 + x_1 x_2 x_3 + x_1 x_2 x_3 x_4 + \dots + \\ &\quad + x_1 x_2 x_3 \dots x_{n-1} + x_1 x_2 x_3 \dots x_n + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n \end{aligned}$$

$$\text{For } i = 1, M_2^n \Big|_{x_1=0} = x_3 + x_4 + \dots + x_n + \bar{x}_2 \bar{x}_3 \dots \bar{x}_n$$

$$\text{For } i = 2, M_2^n \Big|_{x_2=0} = x_1 + x_3 + x_4 + \dots + x_n + \bar{x}_1 \bar{x}_3 \dots \bar{x}_n$$

$$\begin{aligned} \text{For } i > 2, M_2^n \Big|_{x_i=0} &= x_1 + x_3 + x_4 + \dots + x_{i-1} + x_{i+1} + \dots + x_n + \\ &+ x_1x_2 + x_1x_2x_3 + \dots + x_1x_2x_3 \dots x_{i-1} + \\ &+ \bar{x}_1\bar{x}_2\bar{x}_3 \dots \bar{x}_{i-1}\bar{x}_{i+1} \dots \bar{x}_n \end{aligned}$$

Thus biasing a pin to zero "stops" the well-ordered sequence, the length of the longest product being $i-1$. The singleton x_i , of course, does not appear any more, but the product A^{n-1} ($A^{n-1} = \bar{x}_1\bar{x}_2 \dots \bar{x}_{i-1}\bar{x}_{i+1} \dots \bar{x}_n$) of the remaining variables appears in the expression.

Biasing the i^{th} Pin of M_2^n to 1

$$\text{For } i = 1, M_2^n \Big|_{x_1=1} = 1 + x_3 + x_4 + \dots + x_n + x_2 + x_2x_3 + x_2x_3x_4 + \dots + x_2x_3x_4 \dots x_n$$

$$\begin{aligned} i = 2, M_2^n \Big|_{x_2=1} &= x_3 + x_4 + \dots + x_n + x_1x_3 + x_1x_3x_4 + \dots + \\ &+ x_1x_3x_4 \dots x_n \end{aligned}$$

$$\begin{aligned} i > 2, M_2^n \Big|_{x_i=1} &= x_1 + x_3 + x_4 + \dots + x_{i-1} + 1 + x_{i+1} + \dots + x_n + \\ &+ x_1x_2 + x_1x_2x_3 + \dots + x_1x_2x_3 \dots x_{i-2} + \\ &+ x_1x_2x_3 \dots x_{i-2}x_{i-1}x_{i+1} + \dots + x_1x_2x_3 \dots x_{i-1}x_{i+1} \dots x_n \end{aligned}$$

For the two cases when $i = 1$ or 2 , this results in a well ordered sequence which does not have a term missing. Also, all singletons are present when $i = 1$, but when $i = 2$, singleton x_1 is absent. For $i > 2$, the well ordered sequence obtained has a term of length $(i-1)$ absent, and of course the singleton x_i disappears without affecting any other singleton.

Duplicating i^{th} and j^{th} Pins of M_2^n ($j > i$)

$$\begin{aligned} i=1 \text{ and } j=2, M_2^n \Big|_{x_1=x_2} &= x_3 + x_4 + \dots + x_n + x_1x_3 + x_1x_3x_4 + \dots + \\ &+ x_1x_3x_4 \dots x_n + \bar{x}_1\bar{x}_2\bar{x}_3 \dots \bar{x}_n \end{aligned}$$

$$\begin{aligned}
 i=1 \text{ and } j > 2, M_2^n \Big|_{x_1=x_j} &= x_3 + x_4 + \dots + x_{j-1} + x_{j+1} + \dots + x_n + \\
 & x_1 x_2 + x_1 x_2 x_3 + \dots + x_1 x_2 x_3 \dots x_{j-2} + \\
 & x_1 x_2 x_3 \dots x_{j-1} x_{j+1} + x_1 x_2 x_3 \dots x_{j-1} x_{j+1} \dots x_n + \\
 & + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_{j-1} \bar{x}_{j+1} \bar{x}_n
 \end{aligned}$$

$$\begin{aligned}
 i=2 \text{ and } j > 2, M_2^n \Big|_{x_2=x_j} &= x_1 + x_2 + x_3 + x_4 \dots + x_{j-1} + x_{j+1} \dots + x_n + \\
 & x_1 x_2 + x_1 x_2 x_3 + x_1 x_2 x_3 \dots x_{j-2} + \\
 & x_1 x_2 x_3 \dots x_{j-1} x_{j+1} + \dots + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_{j-1} \bar{x}_{j+1} \dots \bar{x}_n
 \end{aligned}$$

$$\begin{aligned}
 i > 2, j > 2; M_2^n \Big|_{x_i=x_j} &= x_1 + x_2 + x_3 + \dots + x_{i-1} + x_{i+1} + \dots + \\
 & + x_{j-1} + x_{j+1} + \dots + x_n + x_1 x_2 + x_1 x_2 x_3 + \\
 & + \dots + x_1 x_2 x_3 \dots x_{j-2} + x_1 x_2 x_3 \dots x_{j-1} x_{j+1} + \\
 & + \dots + x_1 x_2 x_3 \dots x_{j-1} x_{j+1} \dots x_n + \\
 & + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_{j-1} \bar{x}_{j+1} \dots \bar{x}_n
 \end{aligned}$$

This operation is similar to the operation of biasing a pin to 1, except for the product A^{n-1} attached to the resulting function.

3.2 SINGLE-ELEMENT-REALIZABLE FUNCTIONS

Definition: A switching function which can be realized by using only one M_2 module with a finite number of input pins and without using complemented inputs is defined to be a single-element realizable (SER) function of type II.

A single-element realizable function of type I is a well-ordered sequence (w.o.s.), defined in Chapter 1.

Theorem 3.1: All SER type I functions are SER type II.

Proof: From before,

$$\begin{aligned} M_2^n &= 1 + M_1^n + x_1 x_2 x_3 \dots x_n + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n \\ &= 1 + M_1^{n-1} + x_n + x_1 x_2 x_3 \dots x_{n-1} + x_1 x_2 x_3 \dots x_n + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n \end{aligned}$$

Now biasing the n^{th} pin to 1 we have

$$M_2^n \Big|_{x_n=1} = M_1^{n-1}$$

This shows that a M_2 module of n pins with its n^{th} pin connected to the constant 1 is the same as an M_1 module of $(n-1)$ input pins. Therefore, any SER type I function which needs m pins on M_1 can be realized by an M_2 module with at most $m+1$ input pins.

It is easy to see that there are SER type II functions which are not SER type I, since M_2^n itself is not a w.o.s. and therefore is not a SER type I function.

Theorem 3.2: A SER type II function F is either SER type I, or $F + A^D$ is SER type I. (The product A^D contains at least all the variables appearing in the rest of the function in complemented form.)

Proof: In Section 3.1, the three possible operations are shown to result in classes of functions which are either SER type I, or they result in functions that differ from a SER type I by the addition of A^D .

In Ref. 1 it is shown that repeated execution of the three operations on a M_1^n module results only in SER type I functions. Furthermore, SER type I functions when subjected to these operations result only in functions of the same kind.

SER type II functions differ from type I only in the "addition" of a term A^D . When the three operations are applied to type II

functions, either the term A^p vanishes and a type I function remains, or the length of the product A is reduced. Consequently, repeated application of the operations either results in another type II function, or a type I function.

3.3 STANDARDIZATION OF A WELL-ORDERED SEQUENCE

Let W denote a well-ordered sequence.

What follows is the description of a procedure to relabel the variables in a well-ordered sequence (w.o.s.) such that all the functions equivalent under permutation look alike. This will be called the standard form of a w.o.s. As a matter of notation, let subscripted x's denote the variables in this standard form of a well-ordered sequence.

A variable should be renamed x_i if it is the only variable that appears in a product of length i , p_i , and does not appear in p_{i-1} , $i > 2$. In case a set K of more than one variables appears in p_i and not in p_{i-1} , partition K into two subsets, K_1 and K_2 , having the number of elements k_1 and k_2 , respectively, such that $k = k_1 + k_2$ and K_1 contains variables that appear as singletons in W and K_2 contains those which do not appear as singletons in W. Now the variables in K_2 should be labeled from x_i to $x_{i-(k_2-1)}$ (in any order) and variables in K_1 labeled from x_{i-k_2} to $x_{i-(k-1)}$. (Again the order is not important.) The function in the renamed variables gives a standard form for the well-ordered sequence. Now we defined a representation vector RV with $2n$ elements as follows:

$$RV = (v_0, v_1, v_2, \dots, v_n, u_2, u_3, \dots, u_n)$$

$$v_0 = 1 \text{ if the constant } 1 \text{ is present in } W$$

$$= 0 \text{ if the constant } 1 \text{ is not present in } W$$

$$v_j = 1 \text{ if } x_j \text{ is present as a singleton in } W$$

$$v_j = 0 \text{ if } x_j \text{ is not present as a singleton in } W$$

$$u_j = 1 \text{ if } p_j \text{ is present in } W$$

$$= 0 \text{ if } p_j \text{ is not present in } W$$

Lemma 3.1: Two well-ordered sequences W_1 and W_2 have the same representation vector (RV) iff they are equivalent under permutation.

The proof of the lemma follows directly from the construction procedure.

3.4 REALIZATION PROCEDURE

A procedure to realize a given single-element-realizable function of type II by using one module (M_2^P) with unrestricted pin count will now be given, where it is assumed that all input variables are available only in true form, while the output is available in both true and complemented forms.

All the SER type II functions can be written in **one of the** following two forms:

IIa, a "well-ordered sequence", in which case it is a type I function

IIb, " A^n + well-ordered sequence"

We first describe the procedure when the function is of type IIa. We start with the RV of the function and obtain the realization in standard form, from which it is a matter of simple relabeling to get the desired realization.

Let V represent the RV for the function.

$$V = (v_0, v_1, v_2, \dots, v_n, u_2, u_3, \dots, u_n)$$

In case V terminates with a string of 0's and the corresponding v 's are 1's, a vector V_1 , corresponding to V is formed by dropping the terminal u 's and the corresponding v 's. The omitted v 's are taken care of at the end of the algorithm.

Let y 's represent the ordered pin variables. Then an input vector $Y = y_1, y_2, y_3, \dots, y_p$, where p represents the number of pins, will describe the required realization if the y 's are replaced by the variables and constants connected to the corresponding pins.

Start with **the** ordered three-tuple (v_1, v_2, u_2) and make connections on y_1, y_2, y_3 as follows.

v_1	v_2	u_2	y_1	y_2	y_3	Singleton to be EX-ORed at the end
0	0	0	x_1	x_2	x_1	none
0	0	1	x_1	x_2	-	x_1
0	1	0	x_1	x_2	x_2	x_1
0	1	1	x_1	1	x_2	none
1	0	0	x_1	x_2	x_1	x_1
1	0	1	x_1	x_2	-	none
1	1	0	x_1	x_2	x_2	none
1	1	1	1	x_1	x_2	none

Entries marked '-' indicate that the corresponding pin is not used.

Now in general for an ordered pair (v_i, u_i) , after taking care of (v_{i-1}, u_{i-1}) with $j-1$ pins, make the following connections:

$v_i u_i$	$y_j y_{j+1}$	Singleton to be EX-ORed at end
0 0	$x_i x_i$	
0 1	$x_i -$	x_i
1 0	$x_i 1$	
1 1	$x_i -$	

Continue the above substitution until all the products are taken care of. Denote by m the number of pins used.

To take care of the singletons x_i which either were eliminated initially or were required to be "added" (EX-ORed) in accordance with the above tables, proceed as follows. Make a list of those x_i which have to be added. Unless the special case discussed below holds, apply the constant 0 to pin $m+1$ and the x_i on the list to pins $m+2, m+3, \dots$, until all members of the list have been connected. In the special case where the number of singletons in the list is even and the longest product contains all these singletons, pins $m+1$ is not biased to 0 and the x_i on the list are connected to pin $m+1, m+2, \dots$

Finally, a check is made to see if among the pin variables there is at least one biased to 1. (This is required to get the form of type IIa.) If not and there is a pin connected to 0, another pin must be added and biased to 1. If no pin is connected to 1 and no pin is connected to 0, two pins must be added. The first of these is biased to 0 and the second to 1.

For the case of type IIb functions, the term A^n , i.e., $\bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n$, must be maintained in the final output. Therefore no pin can be biased to 1. Now only two operations can be used, i.e., biasing to 0 and duplicating. As biasing to 0 truncates the string of products,

it is useful only for ending this string before taking care of singletons. With only duplication permissible, the algorithm reduces to the following. Starting with the 3-tuple (v_1, v_2, u_2) connect as follows:

v_1	v_2	u_2	y_1	y_2	y_3	Singleton to be EX-ORed at the end
0	0	0	x_1	x_2	x_1	none
0	0	1	x_1	x_2	-	x_1
0	1	0	x_1	x_2	x_2	x_1
0	1	1	x_1	x_1	x_2	none
1	0	0	x_1	x_2	x_1	x_1
1	0	1	x_1	x_2	-	none
1	1	0	x_1	x_2	x_2	none
1	1	1	x_1	x_1	x_2	x_1

For the recursive step,

v_i	u_i	y_j	y_{j+1}	Singleton to be EX-ORed at the end
0	0	x_i	x_i	none
0	1	x_i	-	x_i
1	0	x_i	x_i	x_i
1	1	x_i	-	none

For each variable appearing only in A and nowhere else, two consecutive pins carrying the same variable should appear at the end. As it is assumed that the output is available in both true and complemented form, v_0 is ignored.

To take care of the singletons, x_i , which either were eliminated initially or were required to be 'added' (EX-ORed) in accordance with the above tables, proceed as follows. Let m denote the number of pins

already used. Make a list of those x_i which have to be added. Connect the constant 0 to pin $(m+1)$ and the x_i on the list to pins $m+2, m+3, \dots$, until all the x_i have been taken care of, except when the number of singletons in the list is even and all appear in the longest product. In that special case, the x_i on the list are connected to pins $m+1, m+2, \dots$.

To verify that this algorithm does give the object function, one only needs to check the tables given in the procedure, which are obtained from the operations in Section 3.1. In each case the best alternative is chosen to take care of the singletons.

Theorem 3.3: The above procedure yields the minimum number of pins.

Proof: Reviewing the procedure, it is observed that the first appearance of a variable x_i in a product term is represented by a u_i and a corresponding singleton by v_i . To obtain the function, it is necessary to match both of these elements. Looking at the ordered pair simultaneously, as is done in the procedure, assures that, if possible, both v_i and u_i are simultaneously satisfied without additional pin cost, and only if necessary is an additional pin used at the end to take care of the required v_i . Also, at any step in the procedure if all the remaining u_i are 0, the most economical connection to realize it is used by connecting a single pin to 0. Therefore, the procedure is minimal.

Example: Realize the following SER function by M_2 :

$$F = 1 + x_1 + x_4 + x_5 + x_1x_2 + x_1x_2x_3x_4 + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5$$

Solution: This function is of SER type IIb and is in standard form.

$$\begin{aligned}
 V &= (1; 1,0,0,1,1; 1,0,1,0) \\
 V_1 &= (1; 1,0,0,1; 1,0,1) \\
 (v_1, v_2, u_2) &= (1,0,1) & y_1 &= x_1 \\
 & & y_2 &= x_2 \\
 (v_3, u_3) &= (0,0) \text{ (singleton o.k.)} & y_3 &= x_3 \\
 & & y_4 &= x_3 \\
 (v_4, u_4) &= (1,1) \text{ (singleton o.k.)} & y_5 &= x_4 \\
 \text{for rebuilding } V &\text{ from } V_1 & y_6 &= 0 \\
 & & y_7 &= x_5
 \end{aligned}$$

3.5 REALIZATION OF EX-OR, AND, and OR FUNCTIONS

Note that because it is assumed that the module output is available in both phases, it does not make any difference whether the function or its complement is realized.

Realization of EX-OR of m Variables

Using M_1^{m+1} and biasing the second pin to 0, the object function is realized and this obviously is minimum, as at least one pin would be needed to get any function other than the module.

Using M_2 , $m+2$ pins are needed when complemented inputs are not allowed. Pin y_2 is connected to 0 and y_{m+2} to 1. This solution is obtained by the algorithm and is minimum.

When complemented inputs are available, there also exists a solution that needs $m+1$ pins. Here \bar{x}_1 is connected to pin two.

Realization of 'AND' of m Variables

When complemented inputs are not available, a solution can be obtained by the algorithm of the previous section. This solution needs $2m$ pins and is

$$y_1 = y_3 = x_1$$

$$y_2 = x_2$$

$$y_4 = y_5 = x_3$$

...

$$y_{2m-2} = y_{2m-1} = x_m$$

$$y_{2m} = 1$$

As the algorithm results in a solution with minimum pins, this is a minimum when complemented inputs are not available. In case complemented inputs are allowed, the last two substitutions can be replaced by $y_{2m-2} = x_m$ and $y_{2m-1} = \bar{x}_m$, so that pin $2m$ is not needed.

Using M_1 modules, one needs $(2m-1)$ pins, as is shown by Patt.

Realization of OR of m Variables

OR or NOR cannot be realized by a single M_1 module, because it is a SER type IIb function. By the algorithm of the previous section, the solution for a M_2 module can be obtained in the following manner.

$$F = 1 + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n$$

$$RV = (1, 0, 0, 0, \dots, 0; 0, 0, \dots, 0)$$

No singleton is desired in the function. In accordance with the previous procedure, as a first step the three tuple $(0, 0, 0)$ must be considered and then the rest of the variables appearing only in A^n . The following solution is obtained in this manner.

$$y_1 = y_3 = x_1$$

$$y_2 = x_2$$

$$y_4 = y_5 = x_3$$

⋮
⋮
⋮

$$y_{2m-2} = y_{2m-1} = x_m$$

It is not difficult to see that the solution results in the minimum number of pins required to realize an 'OR', even when complemented inputs are allowed.

3.6 REALIZATION OF M_2^{n-1} AND M_1^{n-1} FROM M_2^n AND M_1^n

To obtain M_2^{n-1} from M_2^n :

$$\begin{aligned} M_2^n &= 1 + M_1^n + x_1 x_2 x_3 \dots x_n + \bar{x}_1 \bar{x}_2 \dots \bar{x}_n \\ &= 1 + M_1^{n-1} + x_n + x_1 x_2 x_3 \dots x_{n-1} + x_1 x_2 x_3 \dots x_n + \bar{x}_1 \bar{x}_2 \dots \bar{x}_n \end{aligned}$$

By biasing the n^{th} pin to zero

$$\begin{aligned} M_2^n \Big|_{x_n=0} &= 1 + M_1^{n-1} + x_1 x_2 x_3 \dots x_{n-1} + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_{n-1} \\ &= M_2^{n-1} \end{aligned}$$

Thus biasing the n^{th} pin of M_2^n to zero results in M_2^{n-1} .

To obtain M_1^{n-1} from M_2^n :

By biasing the n^{th} pin to 1

$$\begin{aligned} M_2^n \Big|_{x_n=1} &= 1 + M_1^{n-1} + 1 + x_1 x_2 x_3 \dots x_{n-1} + x_1 x_2 x_3 \dots x_{n-1} + 0 \\ &= M_1^{n-1} \end{aligned}$$

Thus biasing the n^{th} pin of M_2^n to 1 gives M_1^{n-1} . M_1^{n-1} can also be obtained from M_1^n by biasing the n^{th} pin to zero - but in order to

synthesize M_2^{n-1} from M_1^n one has to go through the synthesis procedure described by Patt, which will need a large number of modules.

3.7 REALIZATION OF M_2^{n+1} AND M_1^{n+1} FROM M_1 AND M_2 MODULES WITH AT MOST n PINS

Recall that

$$\begin{aligned} M_2^{n+1} &= 1 + M_1^{n+1} + x_1 x_2 x_3 \dots x_{n+1} + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_{n+1} \\ &= 1 + M_1^n + x_{n+1} + x_1 x_2 x_3 \dots x_n + x_1 x_2 x_3 \dots x_{n+1} + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_{n+1} \end{aligned}$$

Since

$$M_2^n = 1 + M_1^n + x_1 x_2 x_3 \dots x_n + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n,$$

$$M_1^n = 1 + M_2^n + x_1 x_2 x_3 \dots x_n + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n$$

Substituting this for M_1^n we have

$$\begin{aligned} M_2^{n+1} &= 1 + (1 + M_2^n + x_1 x_2 x_3 \dots x_n + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n) + x_{n+1} + x_1 x_2 x_3 \dots x_n + \\ &\quad + x_1 x_2 x_3 \dots x_{n+1} + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_{n+1} \\ &= M_2^n + x_{n+1} + x_1 x_2 x_3 \dots x_n x_{n+1} + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n (1 + \bar{x}_{n+1}) \\ &= M_2^n + x_{n+1} (1 + x_1 x_2 x_3 \dots x_n + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n) \\ &= M_2^n + x_{n+1} (M_2^n + M_1^n) \end{aligned}$$

Using three modules, M_2^{n+1} can be synthesized from two n -pin modules and one three-pin module as shown in Fig. 3.1.

Now for M_1^{n+1} we know

$$\begin{aligned} M_1^{n+1} &= M_1^n + x_{n+1} + x_1 x_2 x_3 \dots x_n \\ &= 1 + M_2^n + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n + x_1 x_2 x_3 \dots x_n + x_{n+1} + x_1 x_2 x_3 \dots x_n \\ &= 1 + M_2^n + x_{n+1} + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n \end{aligned}$$

$$\text{Since } M_2^n = 1 + M_1^n + x_1 x_2 x_3 \dots x_n + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n$$

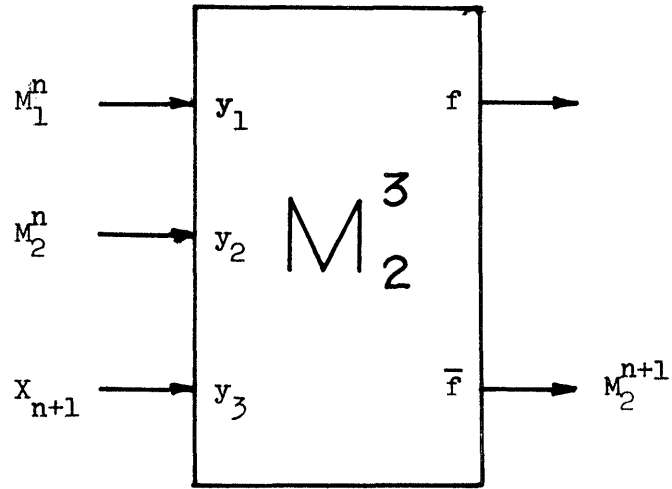


Fig. 3.1 Synthesis of M_2^{n+1}

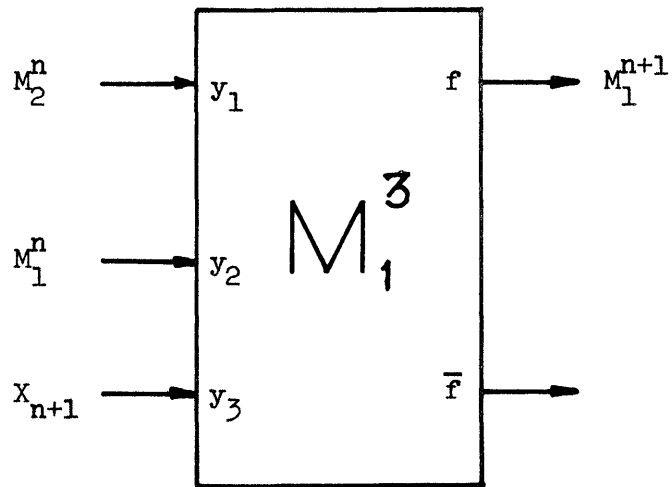


Fig. 3.2 Synthesis of M_1^{n+1}

$$\begin{aligned} M_2^n \cdot M_1^n &= (x_1 x_2 x_3 \dots x_n + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n) M_1^n \\ &= \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n \end{aligned}$$

$$\text{and } M_1^{n+1} = 1 + M_2^n + x_{n+1} + M_2^n \cdot M_1^n$$

Using three modules, one of which has three pins and the others n pins, M_1^{n+1} can be synthesized as shown in Fig. 3.2.

CHAPTER 4
SYNTHESIS BY M_2

In this chapter two synthesis approaches are discussed which are simple from the conceptual point of view, but not at all optimal. A two-level synthesis approach using EX-OR as the second (output) level connective is discussed which is a modification of the approach suggested by Patt. While Patt's procedure is optimal for M_1 , this modified approach is not optimum for M_2 , although the number of M_2 modules used in this approach is always less than or equal to that in Patt's realization. Also, a more or less trivial tree synthesis approach is suggested which makes use of modules with a smaller number of inputs. While in the two-level approach no bound is placed on the number of input pins, in the tree approach the number of pins can be kept as small as three, while the number of levels of logic will depend on the particular function on hand. Throughout, only uncomplemented input variables are used.

4.1 A TWO-LEVEL SYNTHESIS APPROACH

The procedure which follows is a modification of Patt's method. The details of his method are omitted, while the modification is discussed in full.

Step-by-Step Procedure

Step 1. Obtain the unique irredundant ring sum expression (IRSE) of the object function (say F).

Step 2. Define the function $F^* = F + \bar{x}_1\bar{x}_2\bar{x}_3\dots\bar{x}_n$. Also obtain the IRSE of F^* .

Step 2. Prepare minimal M tables for F and F*. The M table, as described by Patt, lists the well-ordered sequences of a function in such a manner that this can finally be modified and minimized to have a minimum number of well-ordered sequences, which when summed mod 2 give the object function. Patt has described an elaborate procedure to obtain and minimize this M table and the minimum is, in general, not unique.

Step 3. Out of the two tables obtained, select the one with the smaller number of well-ordered sequences. This number gives the number of M_2 modules required in the first level of the realization.

Step 4. If the table chosen is that of F and not F*, the procedure reduces to using each M_2 module as an M_1 module and realizing the function as Patt would. But if the table chosen is that of F* with well-ordered sequences w_1, w_2, \dots, w_r , then use one first-level module to realize $A^n + w_i$ for any i, while the rest of the modules can be used to realize the remaining well ordered sequences which are then summed at the second level. The following example illustrates why this method is far from optimum.

Example: $F = x_1x_2x_3 + x_2x_3 + x_1x_4 + x_2x_4 + x_1x_2$

Singletons are ignored, since they can all be taken care of at the second level. It is trivial to see that F has four well-ordered sequences. On the other hand,

$$F^* = x_1x_2x_3x_4 + x_2x_3x_4 + x_1x_3x_4 + x_1x_2x_4 + x_1x_3 + x_3x_4$$

which has three well-ordered sequences, and therefore by using the above procedure one has to use three modules in the first level, while Patt would use four.

We now give a form for F such that it needs only two first-level modules:

$$F = (x_1 x_2 x_3 + x_2 x_3) + (\bar{x}_1 \bar{x}_2 \bar{x}_4 + x_1 + x_2 + x_4)$$

Each of the two functions in parentheses is M_2 realizable, and thus F needs only two first-level modules in the realization. This example suggests that A^n alone is inadequate for a good realization. But we were unable to find a systematic way of choosing proper A^D 's for optimum realization.

4.2 TREE SYNTHESIS APPROACH

According to Shannon's expansion, any function F of n variables can be expanded about any variable x_i as follows:

$$F = \bar{x}_i F_0 + x_i F_1$$

where $F_0 = F \Big|_{x_i=0}$

and $F_1 = F \Big|_{x_i=1}$

Since $\bar{x} = 1 + x$, one can write

$$\begin{aligned} F &= F_0 + x_i (F_1 + F_0) \\ &= F_0 + x_i F_1 + x_i F_0 \end{aligned}$$

This form is M_2^3 realizable with the output complemented and the following input connections

$$y_1 = F_1$$

$$y_2 = F_0$$

$$y_3 = x_i$$

Now a trivial approach would be to use this decomposition in an arbitrary order until the corresponding input functions are reduced

to SER functions. In the following procedure an attempt is made to choose the sequence of variables for decomposition in a reasonable order, but the procedure is not sufficiently complex to yield very good results in all cases.

Let at any node in a decomposition tree the choice be between decomposing about x_i or about x_j . Then x_j should be used before x_i if one or more of the following conditions are satisfied:

Condition 1
$$F_{00}^{x_i x_j} = F_{10}^{x_i x_j} \quad \text{or} \quad F_{01}^{x_i x_j} = F_{11}^{x_i x_j}$$

Condition 2
$$F_{00}^{x_i x_j} = \overline{F_{01}^{x_i x_j}} \quad \text{and} \quad F_{10}^{x_i x_j} = \overline{F_{11}^{x_i x_j}}$$

Condition 3
$$F_{00}^{x_i x_j} = F_{10}^{x_i x_j} = \text{constant or a single variable}$$

or
$$F_{01}^{x_i x_j} = F_{11}^{x_i x_j} = \text{constant or a single variable}$$

Condition 1 implies that either $F_0^{x_j}$ or $F_1^{x_j}$ is independent of the variable x_i . If x_j is used before x_i , one of the function $F_0^{x_i}$ or $F_1^{x_j}$ will not require the variable x_i , thus resulting in a saving of one module.

Condition 2 implies that $F_0^{x_j} = \overline{F_1^{x_j}}$ and since module outputs are assumed to be available in both the true and complemented forms, expanding the function about x_j before x_i will yield a tree with a smaller number of modules.

Condition 3 implies that either $F_0^{x_j}$ or $F_1^{x_j}$ is a constant or a single variable, so that if F is expanded about x_j before x_i one branch will terminate sooner than it would if x_i were to be used before x_j .

Obviously, at any level in the tree two branches should be tied together if they have the identical function or if they are mutually complementary.

At each node, the node function is realized from two sub-functions and one input variable, which are combined in a single M_2^3 module, because the function $F = G + x_1H + x_1G$ is realizable by a single three-pin M_2 module.

It is shown in the next chapter that any three-variable function can be realized using a single M_2 module. Thus the decomposition level need not be reduced below 3-variable functions, provided one is allowed to use 6-pin modules. With these relaxed constraints, an upper bound on the number of modules in the tree is given in the following theorem.

Theorem 4.1: Any n -variable switching function can be realized by using at most $2^{n-2}-1$ modules.

Proof: (By induction) Basis: It is shown in Chapter 5 that any three-variable function can be obtained by a single M_2 module.

Induction: A function of $(n+1)$ variables can be realized by Shannon's expansion, with a single M_2^3 module and two networks each realizing an n -variable function. If the theorem is true for n , then the number of modules for a $(n+1)$ variable function is $2(2^{n-2}-1) + 1 = 2^{(n+1)-2}-1$.

Note that the modules in the first level realize 3-variable functions and need not have more than six pins, as is shown in the next chapter.

An Example to Illustrate the Tree Approach

$$\text{Object function } F = 1 + x_1 + x_3 + x_1x_2 + x_2x_3 + x_3x_4 + x_2x_3x_4 + x_1x_3x_4x_5 + x_1x_2x_3x_4x_5$$

$$F_0^1 = 1 + x_3 + x_2x_3 + x_3x_4 + x_2x_3x_4$$

$$F_1^1 = x_2 + x_3 + x_2x_3 + x_3x_4 + x_3x_4x_5 + x_2x_3x_4 + x_2x_3x_4x_5$$

Now $F_{00}^{x_1x_2} = 1 + x_3 + x_3x_4$

$$F_{01}^{x_1x_2} = 1$$

and $F_{10}^{x_1x_2} = x_3 + x_3x_4 + x_3x_4x_5$

$$F_{11}^{x_1x_2} = 1$$

By Condition 3 of the rules, x_2 should precede x_1

$$F_0^{x_2} = 1 + x_1 + x_3 + x_3x_4 + x_1x_3x_4x_5$$

$$F_1^{x_2} = 1$$

$$F_{01}^{x_2x_1} = x_3 + x_3x_4 + x_3x_4x_5$$

$$F_{00}^{x_2x_1} = 1 + x_3 + x_3x_4$$

Realizing $F_{01}^{x_2x_1}$ and $F_{00}^{x_2x_1}$ with one module each, the complete

realization needs a total of four modules, as shown in Fig. 4.1.

However, this is not an optimal solution. The same function can be written as

$$F = 1 + x_3 + x_2x_3 + x_3x_4 + x_2x_3x_4 + x_1(1 + x_2 + x_3x_4x_5 + x_2x_3x_4x_5)$$

$$\text{Let } g = 1 + x_3 + x_2x_3 + x_3x_4 + x_2x_3x_4$$

$$h = 1 + x_2 + x_3x_4x_5 + x_2x_3x_4x_5$$

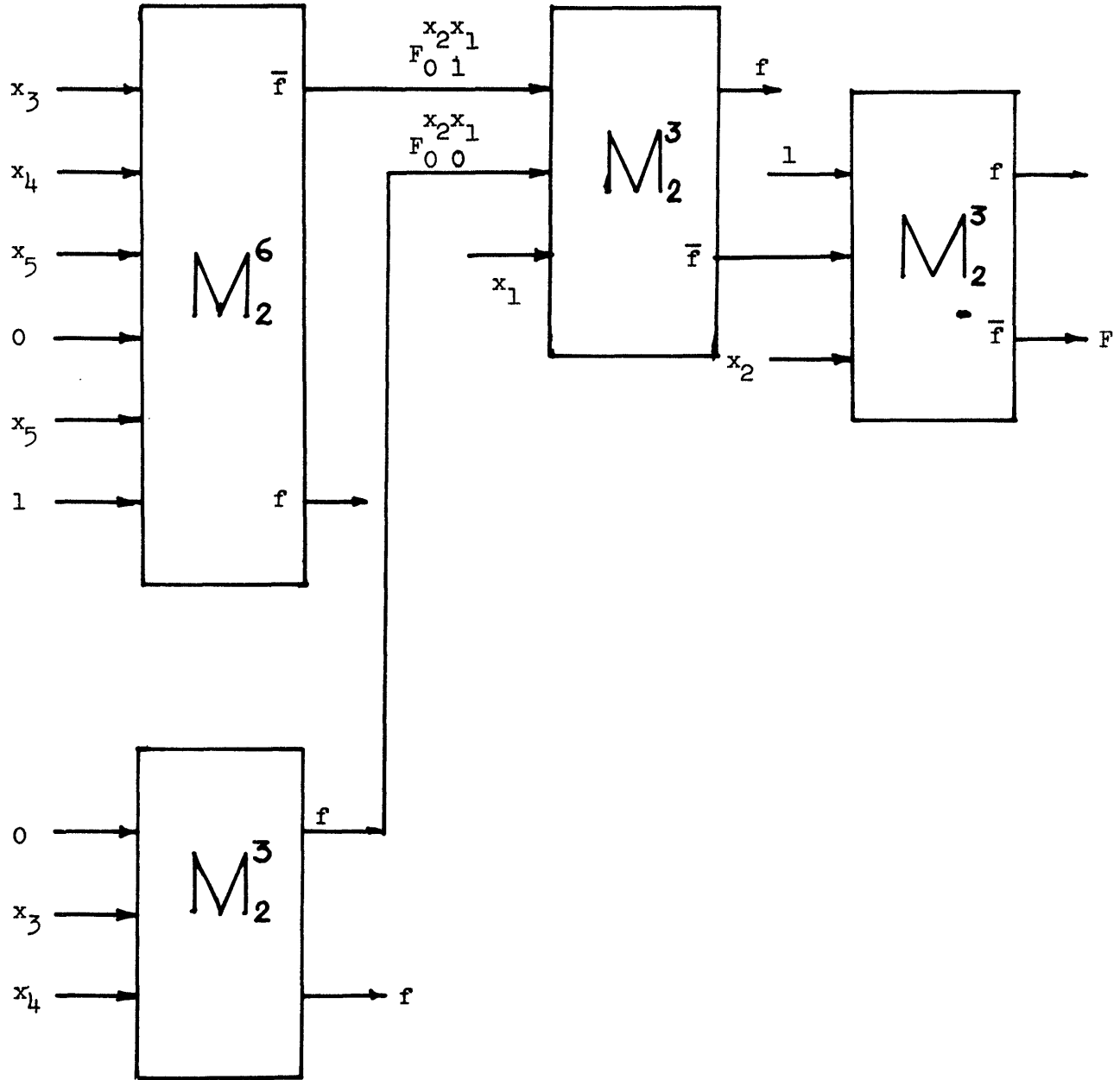


Fig. 4.1 Tree Realization of $F = 1 + x_1 + x_3 + x_1x_2 + x_2x_3 + x_3x_4 + x_2x_3x_4 + x_1x_3x_4x_5 + x_1x_2x_3x_4x_5$

Then g and h are both single-element realizable, since g is a three-variable function and h is SER type I. Then,

$$F = g + x_1 h$$

which can also be implemented by one module. This yields a three-module solution, as shown in Fig. 4.2.

While the two methods presented in this chapter for the synthesis of an arbitrary n -variable switching function are very simple to execute, they are clearly not optimum. In the two-level synthesis the number of M_2 modules used was shown to be less than or equal to the number of M_1 modules used in Patt's optimum synthesis. The upper bound obtained on the number of modules allowing a 6 pin module at the first (input) level is considerably less than the bound obtained by Patt allowing only three-pin modules in a tree-circuit form.

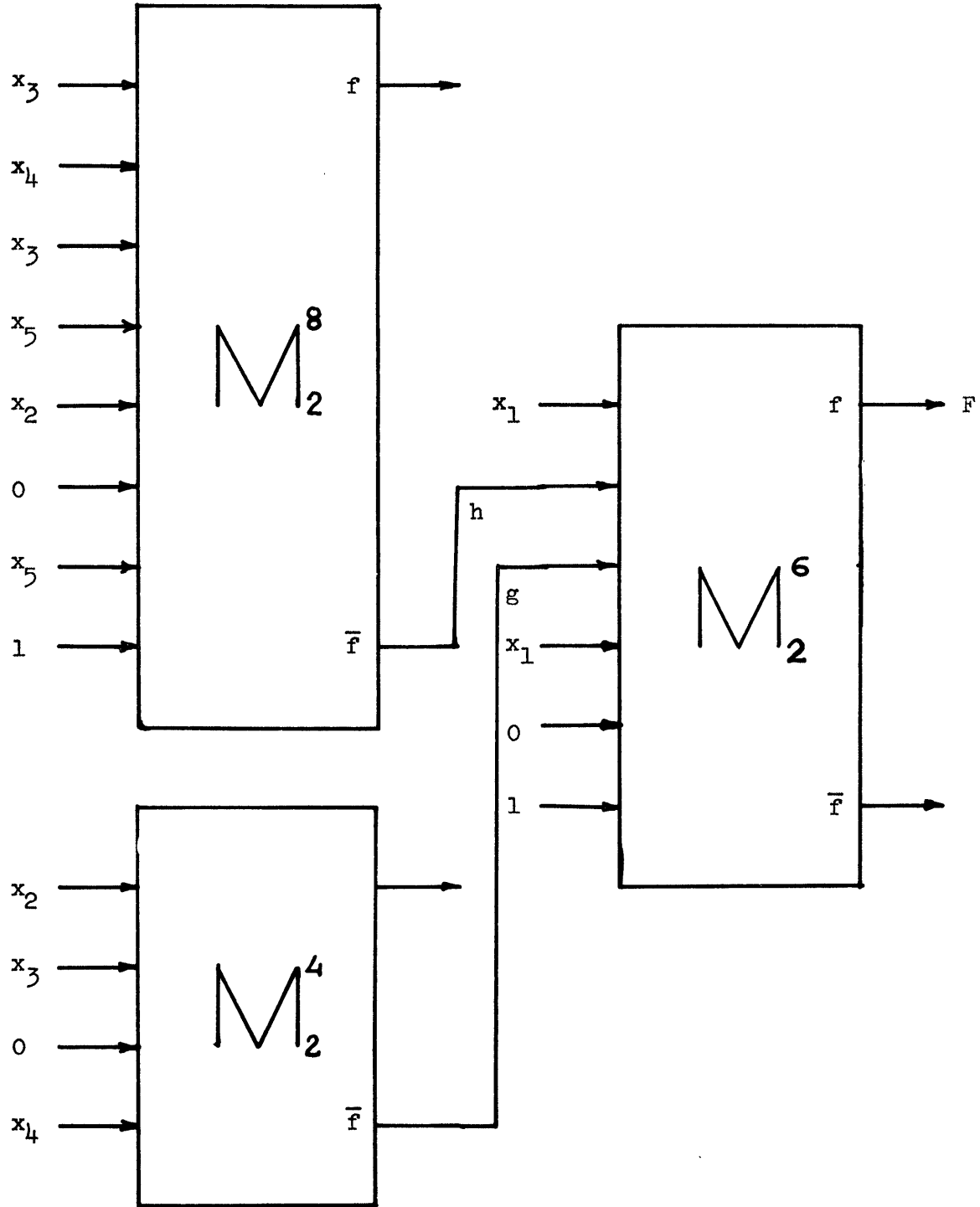


Fig. 4.2 Another Realization of $F = 1 + x_1 + x_3 + x_1x_2 + x_2x_3 + x_3x_4 + x_2x_3x_4 + x_1x_3x_4x_5 + x_1x_2x_3x_4x_5$

CHAPTER 5

SYNTHESIS OF ALL THREE-VARIABLE SWITCHING FUNCTIONS

In this chapter it is shown that all the switching functions of three variables can be realized by using a single 6-pin M_2 module, even when complemented inputs are not available. It is also proved that a M_1 module with unrestricted pin count cannot realize all the functions of three variables, even when all the inputs and the output are available both in true and complemented forms. This shows that M_2 is superior to M_1 , particularly when a small number of variables is involved.

Theorem 5.1: All the three-variable functions are SER-type II functions.

Proof: All the three-variable function $F(A,B,C)$ are SER type II if they can be written as a well-ordered sequence or the ring sum of \overline{ABC} and a well-ordered sequence. It is known from Chapter 3 that singletons (product terms of length 1) in the IRSE do not make any difference in single-element realizability, and also that the absence or presence of product ABC in the IRSE does not change the type of function. Therefore, it is useful to classify all the functions of three variables according to the number of doubletons (products of length 2) in the IRSE. Each class will now be discussed individually.

a) None or one doubleton. The function is clearly SER type IIa.

b) Two doubletons. As there are only three doubletons possible with three variables, there will exist only one doubleton in $F + A^n$. Therefore F is SER type IIb.

c) Three doubletons: Three doubletons in F implies no doubleton in $(F + A^n)$. Therefore F is SER type II.

Hence in all four cases the function is SER type II, which implies that it must be realizable by a single M_2 module, if the output is available in both complemented and uncomplemented form.

Theorem 5.2: The maximum number of pins needed for an n -variable SER function is $3n-2$, given that complemented inputs are not available, and n is even.

Proof: Since the function is SER, it can be written in the standard form and a representation vector (Chapter 3) can be obtained. Let this vector be called V . To get a worst-case function, the last few elements of this vector should not be zero, for otherwise one can, as suggested in the procedure, modify the vector and take care of the singletons at the cost of one pin each, whereas it is known that to obtain a zero in the u 's of V (the right half of V) costs two pins no matter what the corresponding v is (Section 3.2). Therefore, a worst-case function should have all u 's but the last one zero. Now if the function is SER type I, two pins are sufficient to realize $u=0$ and simultaneously $v=0$ or 1 . Therefore, the function should be of SER type IIb and all the v 's should be such that they have to be taken care of at the end, so that for each variable one needs a pin for singletons. Furthermore, for each of the $(n-1)$ products all but the last one costs two pins. Of course, the first three pins are needed to take care of the first two variables.

Consequently, the following are the pin requirements: 3 for u_2 ; $2(n-3)$ for u_3 through u_{n-1} ; 1 for u_n ; 1 for "stopping" the sequence; and $n-1$ for the singletons (v_1 and v_2 together require only one pin). Note that since n is even, $(n-1)$ is odd and a pin is needed for stopping the sequence. Hence the total pin count is $3n-2$, as claimed.

Corollary 5.1: The maximum number of pins needed for an n variable SER function is $3n-3$, given that complemented inputs are not available and n is odd.

Proof: The worst case function is still the same as in Theorem 5.2 but since $(n-1)$ is now even, the pin to stop the sequence is not necessary and the total pin count becomes $3n-3$, as claimed. If only $(n-2)$ out of these $(n-1)$ singletons are to be EX-ORed, the pin to stop the sequence is needed but the case is not worse than the one with $n-1$ singletons.

Theorem 5.2 and the corollary give in fact the least upper bound, because the following class of functions needs that many pins

$$F = x_1 + x_3 + x_4 + \dots + x_{n-1} + x_1 x_2 x_3 \dots x_n + \bar{x}_1 \bar{x}_2 \bar{x}_3 \dots \bar{x}_n$$

Corollary 5.2: The maximum number of pins required to realize any three-variable function by M_2 is six when inputs are not available in complemented form and the output is.

For reference purposes, Table 5.1 lists the pin connections which realize each equivalence class of all the functions of three variables with one M_2 module.

No.	Representative Function	IRSE	Pin Connections					
			1	2	3	4	5	6
1	$\bar{a} \cup \bar{b} \cup \bar{c}$	$1 + ABC$	A	B	A	C	C	1
2	$\bar{a} \cup \bar{b}\bar{c}$	$1 + AB + AC + ABC$	B	C	A	C	-	-
3	$S_{0,1}(a,b,c)$	$1 + AB + BC + CA$	A	B	B	C	-	-
4	$abc \cup \bar{a}\bar{b}\bar{c}$	$1 + A + B + C + AB + BC + CA$	A	B	A	C	0	C
5	$\bar{a}\bar{b} \cup \bar{b}\bar{c} \cup \bar{b}\bar{c}$	$1 + A + AB + BC + CA + ABC$	B	0	C	A	A	-
6	$\bar{a}\bar{c} \cup \bar{a}\bar{b} \cup abc$	$1 + A + BC$	B	C	0	A	B	1
7	$\bar{a} \cup \bar{b}\bar{c} \cup \bar{b}\bar{c}$	$1 + A + AB + AC$	B	C	A	C	A	-
8	$S_{0,1,3}(a,b,c)$	$1 + AB + BC + CA + ABC$	A	0	B	C	-	-
9	$\bar{a}\bar{b} \cup \bar{a}\bar{c}$	$1 + C + AB + AC$	B	C	A	-	-	-
10	$S_{0,2}(a,b,c)$	$1 + A + B + C$	A	0	B	C	1	-

Table 5.1 Realization of All Three-Variable Functions

Note that equivalence classes 1, 4 and 6 are the only ones requiring six pins.

Theorem 5.4: All the functions of three variables cannot be realized by a single M_1 module with any number of pins even allowing input variables both in true and complemented forms as well as output in both forms.

Proof: (By counterexample.) The following three-variable function F cannot be realized by a M_1 module with unrestricted number of pins:

$$F = x_1x_2 + x_1x_3 + x_2x_3$$

By definition, F is not a well-ordered sequence because it has three doubletons. Therefore F cannot be realized by M_1^n with.

uncomplemented inputs. The complemented inputs can be used in one or both of the following two ways:

a) Variable x_i is connected to one pin and \bar{x}_i is connected to another. The effect of this is similar to biasing a pin to zero, because all the products where both x_i and \bar{x}_i appear vanish, as though one of them were zero. But biasing a pin to zero in a M_1 module always results in a w.o.s. and thus can be of no help in realizing the function on hand.

b) Variable x_i appears either in uncomplemented or in complemented form. This means that the function realized can be written as a well-ordered sequence in x_i^* , where x_i^* is either x_i or \bar{x}_i . Now $F = x_1x_2 + x_2x_3 + x_1x_3$ cannot be put into the form of a well-ordered sequence because complementing a variable throughout the expression cannot change the number of doubletons in the new expression.

Since neither way of using complemented variables can realize three doubletons, it follows that F cannot be realized by a single M_1 module.

It happens that for the case of $n=2$, the bound given in Theorem 5.2 can be improved by the use of complemented variables.

Definition: A module is called globally universal for j variables, denoted $G.U.j$, if and only if all the functions of j variables can be realized from this module assuming that complemented inputs and outputs are available.

Theorem 5.3: Module M_2^3 is $G.U. 2$.

Proof: To prove that a module is $G.U.j$, it must be shown that at least one function from each equivalence class of j variables can be

obtained from the module. There are two equivalence classes of two-variable functions. These are listed below with all their members.

(1) $ab; \bar{a}b; a\bar{b}; \bar{a}\bar{b}; a \cup b; \bar{a} \cup b; a \cup \bar{b}; \bar{a} \cup \bar{b}$

(2) $\bar{a}b \cup a\bar{b}; \bar{a}\bar{b} \cup ab$

We will show how one function from each of these classes can be obtained from M_2^3 .

(1) $M_2^3 = 1 + y_2 + y_1y_3 + y_2y_3$

$$\begin{aligned} M_2^3 \Big|_{\substack{y_1=a \\ y_2=y_3=b}} &= 1 + ab = \overline{ab} = \bar{a} \cup \bar{b} \end{aligned}$$

This function belongs to Class 1.

(2) Substituting $y_1 = \bar{y}_2 = a$ and $y_3 = b$

$$\begin{aligned} M_2^3 \Big|_{\substack{y_1=\bar{y}_2=a \\ y_3=b}} &= 1 + \bar{a} + ab + \bar{a}b \\ &= 1 + \bar{a} + b \\ &= a + b = \bar{a}\bar{b} \cup \bar{a}b \end{aligned}$$

This function belongs to Class 2.

CHAPTER 6

SYNTHESIS OF FOUR-VARIABLE FUNCTIONS

Here algorithms for the synthesis of four-variable functions by means of two modules are developed. The key notion is that the full logic power of each module is exploited. As one expects, the procedures that result are not simple, even for four variables. But they are suitable for programming, in which case it is feasible to extend them to a larger number of variables. It is shown that there exist functions of four variables that are not two-module realizable when only uncomplemented variables are available.

Lemma 6.1: All the four-variable functions cannot be realized by a single M_2 module, even when complemented inputs and outputs are available and the number of input pins is unrestricted.

Proof: (By counterexample) The following four-variable function F cannot be realized by a single M_2 module:

$$F = x_1x_2 + x_1x_3 + x_2x_4 + x_2x_3 + x_2x_4 + x_3x_4$$

As pointed out before, with the operations on hand (biasing a pin to a constant and duplication) one can only get functions of SER type II. The function F is clearly not SER type II. This leaves only complemented inputs at our disposal. Again, two possibilities are open:

- a) Connecting a variable x_i on the p th pin and \bar{x}_i on the q th pin. Without lack of generality, let $q > p$. In this case product A^n will be eliminated and the effect is the same as if these connections were made on M_1^n , which can only give a well-ordered sequence.

- b) Connecting only one polarity of each variable, either x_i or \bar{x}_i . If in this case the function were realizable by a M_2 module, then it must be expressible as a SER type II function in some polarity of variables. But it is easy to see that no matter what set of variables is complemented, there remain six doubletons, and no tripletons can be created. So the expression obtained by any set of complementations is not a w.o.s. because of the six doubletons. If the expression were type IIb, then it would have to have three or four tripletons. But it has none. Thus F cannot be put in SER type II form no matter what the polarity of the variables.

Because any function of four variables can be decomposed as $F = G + x_i H$, where G and H are three-variable functions, and because every three-variable function is M_2 realizable, it follows that every four-variable function is three-module realizable. In light of this, it is reasonable to explore what can be done by two modules. In the following section an algorithm is developed to effectively use the cascade of two M_1 modules, given uncomplemented variables only. Later this algorithm is extended to the case of two M_2 modules in cascade.

6.1 CASCADE OF TWO M_1 MODULES

Before starting on the algorithm, several lemmas are given.

Definition: The functions which can be realized by the cascade of two M_1 modules with uncomplemented inputs are called $M_1 M_1$ realizable ($M_1 M_1 R$).

Lemma 6.2: If F is $M_1 M_1 R$, so is $F + x_i$ for all x_i .

Proof: Let F be realized by two modules of which the second (output) module has p pins. By connecting the $(p+1)$ th pin of this module to the constant 0 and the $(p+2)$ th pin to x_1 , the output will be $F+x_1$, since the w.o.s. realized by the second module is terminated by the constant 0 and x_1 then appears only as a singleton.

In other words, this lemma states that one can synthesize an object function by first specifying a network that realizes the desired p -tuples and then taking care of the singletons in the output module.

Lemma 6.3: If F is M_1M_1R , then so is $F + x_1x_2x_3 \dots x_n$, where the n -tuple contains all the input variables.

Proof: Let the first module realize a function f and the second module realize a function ϕ which, when appropriate substitutions are made, yields F .

Suppose the IRSE for ϕ has the term $t = x_1x_2x_3 \dots x_n f$. If f has an odd number of terms, then $t = x_1x_2x_3 \dots x_n$. Then eliminating t , which can always be done, will result in the function $F + x_1x_2x_3 \dots x_n$.

In case ϕ does not have the term t and t is not identically equal to zero (i.e., the number of terms in f is odd), adding this product would achieve the objective.

Now consider the case where $t=0$ because there is an even number of terms in the IRSE of f . Then the IRSE of ϕ will not have term t . To realize $F + x_1x_2x_3 \dots x_n$, first change f to $f' = f + x_1x_2x_3 \dots x_n$, so that f' has an odd number of terms. After f is changed to f' , the function obtained either remains F or it becomes $F + x_1x_2x_3 \dots x_n$, as desired. If the function remains F , adding the term $t' = x_1x_2 \dots x_n f'$ will give $F + x_1x_2x_3 \dots x_n$. Thus in all cases $F + x_1x_2 \dots x_n$ can be realized by the M_1M_1 cascade, if F can be.

Lemma 6.3: In the IRSE of a M_1M_1 realizable function F there can be at most n doubletons, and all but one doubleton must have a common variable.

Proof: Again let f represent the IRSE of the first module and ϕ that of the second module, expressed in terms of f . Since ϕ has to be a well-ordered sequence, doubletons in F can be obtained by one or more of the following ways:

- a) f appears as a singleton in ϕ , contributing at most one doubleton, since f is also a w.o.s.
- b) f appears in the doubleton of ϕ , like $x_i f$. Because f can have at most n singletons, term $x_i f$ can contribute at most $n-1$ doubletons, since all the singletons except x_i in f can give rise to a doubleton in this manner.
- c) f appears in the tripleton $x_i x_j f$ and f has the constant 1. This gives rise to the doubleton $x_i x_j$.
- d) The doubleton $x_i x_j$ in ϕ .

Cases b and d cannot apply simultaneously. If case b holds and it contributes $n-1$ doubletons, case c cannot contribute an additional term. Cases a and b only remain and these can contribute at most n doubletons, of which all but one have a common variable. Furthermore, it is obvious that the doubletons arising from cases c and d have a common variable and so the lemma is proved.

Lemma 6.4: In the IRSE of a M_1M_1 realizable function F there can be at most n tripletons. Of these all but two have two variables, x_i and x_j , in common. The two remaining tripletons are of the form $x_i vw$ (vw is a doubleton from f) and uvw (tripleton from f).

Proof: The tripletons in the output function can come about in one of the following ways:

- a) When f , the output of the first module, appears as a singleton in ϕ , it can contribute its tripleton uvw to F .
- b) When f appears in the doubleton of ϕ , like $x_i f$, the doubleton in f , when multiplied by x_i , yields a tripleton.
- c) When f appears in the tripleton in ϕ , like $x_i x_j f$, the singletons in f can give up to $n-2$ tripletons.
- d) When f appears in ϕ as $x_i x_j x_k f$ and f contains the constant 1, the tripleton $x_i x_j x_k$ results, which is one of the possible $n-2$ in case c.
- e) A tripleton in ϕ not involving f .

If case e holds, only one other tripleton (from case a) can be obtained. The maximum number of tripletons from cases a through d is n , since the one tripleton from case d is also one of those from case c. The tripletons from cases c through e have two variables in common and one of these variables also appears in the tripleton from case b.

Lemma 6.5: The realization of a $M_1 M_1 R$ function is not necessarily unique.

Proof: (By counterexample) Either $f = x_1 + x_1 x_3 + x_1 x_3 x_4$ or $f = 1 + x_1 x_3 + x_1 x_3 x_4$ together with $\phi = x_1 f + x_1 x_2 f + x_1 x_2 x_4 f$ give the same function $F = x_1 + x_1 x_3 + x_1 x_2 + x_1 x_3 x_4 + x_1 x_2 x_3 + x_1 x_2 x_4 + x_1 x_2 x_3 x_4$.

We are now ready to give a step-by-step procedure to obtain a $M_1 M_1$ cascade realization of a four-variable function, if it exists. Complemented inputs are assumed not available.

Step 1: Obtain the IRSE of the function and obtain from this a modified function F_m by dropping all singletons and the fourtuple. From Lemmas 6.1 and 6.2 it is clear that if a realization of F_m is found, then a realization of F can be obtained.

Step 2: Check if the doubletons and tripletons satisfy the conditions of Lemmas 6.3 and 6.4, respectively. If not, the function is not M_1M_1 realizable.

Step 3: List all variables x_c which are common to all but at most one of the doubletons, and all those pairs of variables x_i, x_j which are common to all but at most two tripletons.

Step 4: Construct a set P of ordered pairs whose first element is one of the pairs x_i, x_j obtained in Step 3 and whose second element is one of the two elements in this pair (either x_i or x_j) which is also one of the x_c variables obtained in Step 3. This set P may be empty, indicating that the function cannot be realized as a M_1M_1 cascade.

Step 5: Pick an ordered pair from $P(x_p x_q, x_p)$ and rewrite the IRSE of F_m as follows. In the expression for F_m , collect all those terms which contain the second element in the pair, x_p , and not both x_p and x_q . Write these terms in row 2 with x_p factored out, i.e., in the form $x_p(t_1 + t_2 + \dots)$. We call x_p the row factor of row 2, RF_2 , and $t_1 + t_2 + \dots$ is called row 2. Collect all those terms which contain the first element in the ordered pair. Write these terms in row 3 with $x_p x_q$ factored out, i.e., in the form $x_p x_q(t_i + t_j + \dots)$. We call $x_p x_q$ the row factor of row 3, RF_3 , and $t_i + t_j + \dots$ is called row 3.

If there exists a doubleton or a tripleton which does not have x_p , then these terms are placed in row 1. The elements in row 1 must come from f , the output of the first module.

Step 6: Two rows are said to be compatible if they are identical or one is empty. Make the rows compatible according to the following rules:

1. Any singleton or the constant 1 can be added to row 1.
2. The constant 1 can be added to row 2.
3. Any ring sum of products S can be added to row i if $(RF_i)S$ is 0 or the n -tuple $x_1x_2x_3\cdots x_n$.
4. If row j has x_{j+1} as a singleton and x_{j+1} also appears as a variable in row factor $RF(j+1)$, then x_{j+1} can be dropped from row j if the constant 1 is added to row $j+1$.

Only if the application of the above four rules makes all three rows compatible can the function be M_1M_1R . Call G the sum of all the rows when multiplied by the row factors. (a) If G equals F_m or a function differing from F_m by only singletons or the product $x_1x_2x_3\cdots x_n$, then the expression in a non-zero row specifies the function f . If f is a w.o.s. and the row factors are a w.o.s., then F_m is M_1M_1R . (b) If G also differs from F_m by a tripleton $x_p x_q x_r$ and a product of $x_p x_q x_r$ and a non-zero row equals $x_p x_q x_r$ plus possibly the four-tuple $x_1x_2x_3x_4$, then row 4 with $RF^4 = x_p x_q x_r$ and the non-zero row is added. Again the non-zero row specifies the function f and the condition on M_1M_1 realizability is the same as in case (a).

Step 7: Return to Step 5 until a realization is obtained or all pairs in P are exhausted.

Example 6.1:

Step 1: $F_m = x_2x_3 + x_3x_4 + x_1x_2x_3 + x_1x_2x_4 + x_1x_3x_4$

Step 2: Doubletons and tripletons satisfy the conditions of Lemmas 6.3 and 6.4.

Steps 3 and 4: Set P contains all the possible pairs.

$$P = \{(x_1x_2, x_1), (x_1x_2, x_2), \dots \dots \}$$

Step 5: Making a first arbitrary choice, (x_1x_2, x_2)

$$\begin{pmatrix} x_3x_4 + x_1x_3x_4 \\ x_2(x_3 + \dots) \\ x_1x_2(x_3 + x_4 + \dots) \end{pmatrix}$$

To make the rows compatible in accordance with Step 6, doubleton x_3x_4 , when added to second row, generates tripleton $x_2x_3x_4$ which cannot be taken care of with $RF_3 = x_1x_2$. Therefore, the rows cannot be made compatible.

By taking as second choice (x_1x_3, x_3) , F_m can be rewritten as follows.

$$\begin{pmatrix} + x_1x_2x_4 \\ x_3(x_2 + x_4 + \dots) \\ x_3x_1(x_2 + x_4 + \dots) \\ (x_2 + x_4 + x_1x_2x_4) \\ x_3(x_2 + x_4 + x_1x_2x_4) \\ x_3x_1(x_2 + x_4 + x_1x_2x_4) \end{pmatrix}$$

In Step 6 we obtain

Making $f = x_2 + x_4 + x_1x_2x_4$ and $\phi = f + x_3f + x_3x_1f$ a function G

differing from F_m only in singletons is obtained, which can be modified to obtain the object function.

Example 6.2: $F_m = x_1x_2 + x_1x_3 + x_1x_4 + x_1x_2x_3 + x_1x_2x_4$

From steps 2, 3 and 4, $P = \{(x_1x_2, x_1), (x_1x_3, x_1), (x_1x_4, x_1)\}$

Step 5: Selecting the pair (x_1x_2, x_1) , F_m can be rewritten as follows.

$$x_1(1 + x_3 + x_4) \\ x_1x_2(1 + x_3 + x_4)$$

Step 6:

$$x_1(1 + x_3 + x_4) \\ x_1x_2(1 + x_3 + x_4)$$

So $f = 1 + x_3 + x_4$ and $\phi = x_1f + x_1x_2f$ gives a function differing from F_m only in the singleton x_1 .

Theorem 6.1: All the functions of four variables cannot be realized by an M_1M_1 cascade even when complemented inputs are available.

Proof: (By counterexample) By Lemma 6.3, the following function F cannot be realized by a M_1M_1 cascade without complemented variables:

$$F = y_1y_2 + y_1y_3 + y_1y_4 + y_2y_3 + y_2y_4 + y_3y_4$$

Observe that no matter how independent variables are complemented, there remain six doubletons in the expression for F and there will be no tripleton. In the M_1M_1 cascade form assumed, the independent variables connected to the output module can be chosen from y_i or \bar{y}_i arbitrarily, but as was observed before, a variable will never appear both as y_i and \bar{y}_i . For any choice between y_i and \bar{y}_i , we can rewrite F in terms of these choices, and still be left with a function which has

precisely six doubletons and possibly some singletons. Since we know that singletons do not affect realizability, the problem on hand is to realize the related function

$$F_m = x_1x_2 + x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 + x_3x_4$$

where $x_i = y_i$ or \bar{y}_i , as chosen for the output module. Now ϕ can take any one of the following forms, where $c_i = 1$ or 0 :

- a. $c_0 + c_1x_2 + c_3x_3 + c_4c_4 + c_5f + c_6x_1x_2 + c_7x_1x_2x_3 + c_8x_1x_2x_3x_4 +$
 $+ c_9x_1x_2x_3x_4f$
- b. ----singletons---- $c_6x_1x_2 + c_7x_1x_2x_3 + c_8x_1x_2x_3f + c_9x_1x_2x_3fx_4$
- c. " $c_6x_1x_2 + c_7x_1x_2f + c_8x_1x_2fx_3 + c_9x_1x_2fx_3x_4$
- d. " $c_6x_1f + c_7x_1fx_2 + c_8x_1fx_2x_3 + c_9x_1fx_2x_3x_4$

In each of the first three forms of ϕ above, the product terms generate at most one doubleton. So five or more doubletons must be obtained from f , and c_5 must be made 1. The only way one can obtain five or more doubletons from a w.o.s. is by complementing all the variables in the four-tuple $x_1x_2x_3x_4$. But then three tripletions are also generated and these cannot be cancelled. Therefore, none of the three forms can be used.

Now in form d there is the possibility of obtaining three doubletons from x_1f . The remaining three should be obtained from c_5f . But no matter how this is done, these doubletons will each generate a tripletion in x_1f which cannot be cancelled.

Thus all four forms cannot realize F_m , and therefore F is not M_1M_1R even with complemented inputs.

6.2 CASCADE OF TWO M_2 MODULES

The algorithm in the previous section can be extended to handle M_2 modules in a two-module cascade. As before, we let f denote the function realized by the first module and ϕ the function realized by the second module expressed in terms of f . A function realized by a M_2M_2 cascade belongs to one of the following four classes:

- a) f is of SER type IIa and ϕ is of SER type IIa
- b) f is of SER type IIb and ϕ is of SER type IIa
- c) f is of SER type IIa and ϕ is of SER type IIb
- d) f is of SER type IIb and ϕ is of SER type IIb

Each of these classes will be individually discussed. Before developing the algorithm, the following lemma will be helpful.

Lemma 6.6: If F is M_1M_1R , then $F + \bar{x}_1\bar{x}_2\bar{x}_3\dots\bar{x}_n$ is M_2M_2R .

Proof: Let F be realized by an M_1M_1 cascade. Then $f + \bar{x}_1\bar{x}_2\bar{x}_3\dots\bar{x}_n$ can be realized by an M_2M_2 cascade as follows:

If f appears as a singleton in ϕ , modify f to $f_m = f + \bar{x}_1\bar{x}_2\bar{x}_3\dots\bar{x}_n$ and the same ϕ will realize $F + \bar{x}_1\bar{x}_2\bar{x}_3\dots\bar{x}_n$. But if f does not appear as a singleton in ϕ , add a term $t = \bar{x}_1\bar{x}_2\bar{x}_3\dots\bar{x}_n\bar{f}$ in ϕ . Term t can either be zero or $\bar{x}_1\bar{x}_2\bar{x}_3\dots\bar{x}_n$. In the latter case, ϕ will realize $F + \bar{x}_1\bar{x}_2\dots\bar{x}_n$ with the original f and we are done. In case t is zero, use f_m as output from the first module so that t becomes $\bar{x}_1\bar{x}_2\bar{x}_3\dots\bar{x}_n\bar{f}_m$. Since $\bar{x}_1\bar{x}_2\bar{x}_3\dots\bar{f}$ is zero, the modified t has to be equal to $\bar{x}_1\bar{x}_2\bar{x}_3\dots\bar{x}_n$ and the function $F + \bar{x}_1\bar{x}_2\bar{x}_3\dots\bar{x}_n$ will be realized.

Now each one of the four cases listed above will be discussed.

Case a. Same as the M_1M_1 cascade.

Case b. In general, ϕ has to be in one of the following forms:

$$\begin{aligned} \text{b-1} \quad & a_0 + a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 + a_5f + a_6x_1x_2 + a_7x_1x_2x_3 + \\ & + a_8x_1x_2x_3x_4 + a_9x_1x_2x_3x_4f \end{aligned}$$

$$\begin{aligned} \text{b-2} \quad & \text{----singletons----} \quad a_5f + a_6x_1x_2 + a_7x_1x_2x_3 + a_8x_1x_2x_3f + \\ & + a_9x_1x_2x_3fx_4 \end{aligned}$$

$$\begin{aligned} \text{b-3} \quad & \text{----singletons----} \quad a_5f + a_6x_1x_2 + a_7x_1x_2f + a_8x_1x_2fx_3 + \\ & + a_9x_1x_2fx_3x_4 \end{aligned}$$

$$\begin{aligned} \text{b-4} \quad & \text{----singletons----} \quad a_5f + a_6x_1f + a_7x_1fx_2 + a_8x_1fx_2x_3 + \\ & + a_9x_1fx_2x_3x_4 \end{aligned}$$

Since f is SER type IIb, it has to be a function of at least three variables, because all two-variable functions can be classified as SER IIa. If f is a function of four variables, A^4 in f will be $\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4$, and because of Lemma 6.6, $F + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4$ is then M_1M_1 realizable. We will henceforth ignore $\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4$ because as a first step in the procedure functions F and $F + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4$ will be checked for M_1M_1 realizability. Now if f is a three-variable function, say $f(x_1, x_2, x_3)$, $f + \bar{x}_1\bar{x}_2\bar{x}_3$ is SER type IIa. In cases b-1, b-2, b-3, if $a_5 = 0$, the term $A^3 = \bar{x}_1\bar{x}_2\bar{x}_3$ makes no contribution and the function must belong to Case a. But if $a_5 = 1$, then A^3 will be contributed, since the remaining products with A^3 will vanish. In b-4 if $a_5 = 0$, the only contribution which A^3 could make is $\bar{x}_1\bar{x}_2\bar{x}_3x_4$, while if $a_5 = 1$ the contribution has to be either $\bar{x}_1\bar{x}_2\bar{x}_3$ or $\bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2\bar{x}_3x_4$, which is $\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4$. Thus, to take care of these cases, $F + A^3$ for all possible A^3 's should be tested for M_1M_1 realizability and if $F + A^3$ is M_1M_1R , F may be M_2M_1R . Function f in the M_1M_1 cascade for $F + A^3$ must have the same three variables as A^3 and f must also appear

as a singleton in ϕ . Furthermore, related functions of the form $F + \bar{x}_i \bar{x}_j \bar{x}_k x_m$ should be tested for $M_1 M_1$ realizability. If a related function of this form is $M_1 M_1 R$, $a_5 = 0$, and $x_i x_j x_k$ are the variables of f , then F is $M_2 M_1 R$.

Case c: Here there will exist terms like $\bar{x}_i \bar{x}_j \bar{f}$ or $\bar{x}_i \bar{x}_j \bar{x}_k \bar{f}$ in ϕ , but the term $\bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4 \bar{f}$ need not be considered, as it will be taken care of in the beginning.

Observe that in this case all the singletons cannot be taken care of except those appearing as independent variables in ϕ . Since ϕ is SER type IIb, it has to have at least two independent variables and f as its arguments, so that only three cases can arise:

- c-1 ϕ is a three-argument function
- c-2 ϕ is a four-argument function
- c-3 ϕ is a five-argument function

For c-1, the decomposition technique given at the end of Appendix A can be used.

For c-3, as already discussed, $F + \bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4$ only need be tested for $M_1 M_1$ realizability.

For c-2, one term in ϕ is $\bar{x}_i \bar{x}_j \bar{x}_k \bar{f}$. Since f is SER type IIa, the result of this product can be $\bar{x}_i \bar{x}_j \bar{x}_k$ or $\bar{x}_i \bar{x}_j \bar{x}_k x_m$ or $\bar{x}_i \bar{x}_j \bar{x}_k \bar{x}_m$. The problem reduces to trying $F + \bar{x}_i \bar{x}_j \bar{x}_k$ and $F + \bar{x}_i \bar{x}_j \bar{x}_k x_m$ for $M_1 M_1$ realizability such that ϕ uses the variables x_i , x_j , and x_k .

Case d: Here f must be a function of at least three variables and ϕ has to have at least two independent variables along with f . In case ϕ has only two independent variables, the decomposition procedure of Appendix A is applicable. When ϕ has four independent variables, then

$F + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4$ is M_2M_1 realizable and therefore need not be considered.

Similarly, we need not consider the cases where f is a function of four variables. This leaves only the case where ϕ and f both are functions of three independent variables, so that $A_\phi = \bar{x}_i\bar{x}_j\bar{x}_k\bar{f}$. The following are the general forms for ϕ .

$$\begin{aligned} \text{d-1} \quad & a_0 + a_1x_i + a_2x_j + a_3x_k + a_4f + a_5x_ix_j + a_6x_ix_jx_k + a_7x_ix_jx_kf + \\ & + \bar{x}_i\bar{x}_j\bar{x}_k\bar{f} \end{aligned}$$

$$\begin{aligned} \text{d-2} \quad & \text{-----singletons-----} \quad a_4f + a_5x_ix_j + a_6x_ix_jf + a_7x_ix_jfx_k + \\ & + \bar{x}_i\bar{x}_j\bar{x}_k\bar{f} \end{aligned}$$

$$\begin{aligned} \text{d-3} \quad & \text{-----singletons-----} \quad a_4f + a_5x_if + a_6x_ifx_j + a_7x_ifzx_k + \\ & + \bar{x}_i\bar{x}_j\bar{x}_k\bar{f} \end{aligned}$$

In case d-3, $F + \bar{x}_i\bar{x}_j\bar{x}_k$ will be decomposable as an AND of two three-variable functions when the singletons are ignored. This decomposition can be obtained as in Appendix A.

For cases d-1 and d-2 when $a_4 = 0$ the same functions as in case c need be tried for M_1M_1 realizability, and checked to see if the appropriate A's can be added later to obtain the object function by a M_2M_2 cascade. In the two cases where $a_4 = 1$, the tripleton A_f in f is also contributed to F , and the result of the term $t = \bar{x}_i\bar{x}_j\bar{x}_k\bar{f}$ can either be $\bar{x}_i\bar{x}_j\bar{x}_k\bar{x}_m$ or $\bar{x}_i\bar{x}_j\bar{x}_kx_m$ or $\bar{x}_i\bar{x}_j\bar{x}_k$. When $t = \bar{x}_i\bar{x}_j\bar{x}_k\bar{x}_m$ the combined effect of A_f and t is a product like $\bar{x}_i\bar{x}_j\bar{x}_kx_m$ and is one of the functions considered in case c. When $t = \bar{x}_i\bar{x}_j\bar{x}_k$ or $\bar{x}_i\bar{x}_j\bar{x}_kx_m$, functions of the form $F + \bar{x}_i\bar{x}_j\bar{x}_k + \bar{x}_i\bar{x}_j\bar{x}_k\bar{x}_m$ and $F + \bar{x}_i\bar{x}_j\bar{x}_k + \bar{x}_i\bar{x}_j\bar{x}_kx_m$ must be tried for M_1M_1 realizability and then checked to see if appropriate A's can be added to obtain the object function.

In light of the above discussion of all the possible circuit forms, the following algorithm is valid for M_2M_2 synthesis. The procedure is stopped at the end of any step where success is met; otherwise the next step is taken.

Step 1: Try F for M_1M_1 realizability.

Step 2: Try $F_m = F + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4$ for M_1M_1 realizability. If it can be done, obtain the corresponding M_2M_2 realization by the steps in the proof of Lemma 6.6.

Step 3:

- a) Obtain a M_1M_1 realization of $F + \bar{x}_i\bar{x}_j\bar{x}_k$ in which either f or ϕ have $x_i, x_j,$ and x_k as their arguments.
- b) If x_i, x_j, x_k are the arguments of f (ϕ), make f (ϕ) a SER type IIb function and check if $f + \bar{x}_i\bar{x}_j\bar{x}_k$ substituted in ϕ ($\phi + \bar{x}_i\bar{x}_j\bar{x}_k$) gives F .
- c) If x_i, x_j, x_k are the arguments of both f and ϕ , make both f and ϕ SER type IIb and check if F results.
- d) Repeat b and c for a different M_1M_1 realization of $F + \bar{x}_i\bar{x}_j\bar{x}_k$ until all realizations are exhausted.
- e) Repeat a through d for all possible x_i, x_j, x_k .

Step 4: Try $F + \bar{x}_i\bar{x}_j\bar{x}_k x_m$ for a M_1M_1 realization and see if it can be modified to a M_2M_2 realization of F in a manner similar to that of Step 3. Do this for all $\bar{x}_i\bar{x}_j\bar{x}_k x_m$.

Step 5: Decompose, as in Appendix A, the function F such that $F = \phi(x_i, x_j, f)$ and f is SER. F can then be realized, since ϕ is a function of three variables.

Step 6: Decompose $F + \bar{x}_i \bar{x}_j \bar{x}_k$ in the form of an AND of two three-variable functions as shown in Appendix A. If this decomposition exists and one of the subfunctions has x_i , x_j , and x_k as its arguments, obtain the $M_2 M_2$ realization as follows.

$$\text{Let} \quad F + \bar{x}_i \bar{x}_j \bar{x}_k = g(x_i, x_j, x_k) \cdot h(x_i, x_k, x_m)$$

$$\begin{aligned} \text{Then} \quad F &= g \cdot h + \bar{x}_i \bar{x}_j \bar{x}_k \\ &= g \cdot h + \bar{x}_i \bar{x}_j \bar{x}_k h + \bar{x}_i \bar{x}_j \bar{x}_k \bar{h} \\ &= (g + \bar{x}_i \bar{x}_j \bar{x}_k) h + \bar{x}_i \bar{x}_j \bar{x}_k \bar{h} \end{aligned}$$

For F to be realizable by a $M_2 M_2$ cascade, the expression $(g + \bar{x}_i \bar{x}_j \bar{x}_k)h$ has to be a w.o.s. in the arguments x_i , x_j , x_k , and h . If this is true, f can be set equal to h and ϕ is now SER type IIb. Therefore F can be realized in the above form.

Step 7: Now functions of the form $F + \bar{x}_i \bar{x}_j \bar{x}_k + \bar{x}_i \bar{x}_j \bar{x}_m$ and $F + \bar{x}_i \bar{x}_j \bar{x}_k + \bar{x}_i \bar{x}_j \bar{x}_m x_k$ should be tried for $M_1 M_1$ realizability and checked to see if appropriate Λ 's can be added to obtain the object function.

Example 6.3:

$$F = x_1 x_4 + x_1 x_3 + x_2 x_4 + x_2 x_3 + x_1 x_2 x_3 + x_2 x_3 x_4$$

Step 1: Since there is no variable common to three of the four doubletons, F is not $M_1 M_1 R$.

$$\text{Step 2: } F + \bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4 = x_1 x_2 + x_3 x_4 + x_1 x_3 x_4 + x_1 x_2 x_4 + x_1 x_2 x_3 x_4$$

This function is $M_1 M_1 R$:

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

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

To obtain F, we modify f to

$$f_m = x_1x_2 + x_1x_2x_4 + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4$$

which is SER type IIb. Thus F is M_2M_1R .

Theorem 6.2: All the four-variable functions are not M_2M_2R when complemented inputs are not allowed.

Proof: (By counterexample) The following function is not M_2M_2R

$$F = x_1x_2 + x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 + x_3x_4$$

Steps 1 and 2: F and $F + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4$ are not M_1M_1 realizable.

Step 3: Since F is symmetric in all four variables, only one of the functions $F + \bar{x}_1\bar{x}_j\bar{x}_k$ need be tried.

$$F + \bar{x}_1\bar{x}_2\bar{x}_3 = x_1x_4 + x_2x_4 + x_3x_4 + x_1x_2x_3 + x_1 + x_2 + x_3$$

which has the M_1M_1 realization

$$\begin{aligned} &(x_1 + x_2 + x_3 + x_1x_2x_3) \\ &x_4(x_1 + x_2 + x_3 + x_1x_2x_3) \end{aligned}$$

But to realize F, the 4-tuple $x_1x_2x_3x_4$ must be taken care of without making the row or the row factors a function of four variables. But this cannot be done. Hence step 3 fails.

Step 4: Again, because of symmetry, trying $F + \bar{x}_1\bar{x}_2\bar{x}_3x_4$ is enough.

$$F + \bar{x}_1\bar{x}_2\bar{x}_3x_4 = x_1x_2x_4 + x_1x_3x_4 + x_2x_3x_4 + x_4 + x_1x_2x_3x_4 + x_1x_2 + x_1x_3 + x_2x_3$$

It turns out that this expression is not M_1M_1R .

$$\text{Step 5: } F = x_3x_4 + x_1(x_3 + x_4) + x_2(x_3 + x_4) + x_1x_2(1)$$

$$p = x_3x_4$$

$$q = x_3 + x_4$$

$$r = x_3 + x_4$$

$$s = 1$$

For a decomposition such that $F = \phi(x_1, x_2, h(x_1, x_3, x_4))$, the first condition in Appendix A is $p = r^*$, which is not satisfied.

Now we try the decomposition $F = \phi(x_1, x_2, f)$. Here f can be a function of four variables. The condition for a non-trivial decomposition is $q \cup r \cup s \neq 1$, which is also not satisfied.

$$\begin{aligned} \text{Step 6: } F + \bar{x}_1 \bar{x}_2 \bar{x}_3 &= x_1 x_4 + x_2 x_4 + x_3 x_4 + x_1 x_2 x_3 + x_1 + x_2 + x_3 \\ &= x_2 + x_3 + x_1(1 + x_2 x_3) + x_4(x_2 + x_3) + \\ &\quad + x_1 x_4(1) \end{aligned}$$

$$p_1 = x_2 + x_3$$

$$q_1 = 1 + x_2 x_3$$

$$r_1 = x_2 + x_3$$

$$s_1 = 1$$

The condition for the decomposition into an AND of two functions is $ps = qr$, which obviously is satisfied. If $F = g(x_1, x_2, x_3) \cdot h(x_2, x_3, x_4)$, the equations to be solved are.

$$g_1 \cdot h_1 = x_2 + x_3$$

$$g_2 \cdot h_1 = 1 + x_2 x_3$$

$$g_1 \cdot h_2 = x_2 + x_3$$

$$g_2 \cdot h_2 = 1$$

The unique solution is $g_2 = h_2 = 1$, $h_1 = 1 + x_2 x_3$, and $g_1 = x_2 + x_3$.

Then

$$g = g_1 + x_1 g_2 = x_2 + x_3 + x_1,$$

$$g + \bar{x}_1 \bar{x}_2 \bar{x}_3 = 1 + x_1 x_2 + x_1 x_3 + x_2 x_3 + x_1 x_2 x_3,$$

and $(g + \bar{x}_1 \bar{x}_2 \bar{x}_3)h$ is obviously not a w.o.s. so that this step also does not give a solution.

$$\begin{aligned}\text{Step 7: } F + \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2\bar{x}_4 \\ = x_1x_2 + x_3x_4 + x_1x_2x_3 + x_1x_2x_4\end{aligned}$$

which has a M_1M_1 realization as follows.

$$\begin{aligned}(1 + x_3 + x_4 + x_3x_4) \\ x_1x_2(1 + x_3 + x_4 + x_3x_4)\end{aligned}$$

Again to take care of $x_1x_2x_3x_4$, it is necessary to make f or ϕ a 4-variable function, which results in a situation making it impossible to obtain $\bar{x}_1\bar{x}_2\bar{x}_3$ and $\bar{x}_1\bar{x}_2\bar{x}_4$.

$$\begin{aligned}\text{Next we try } F + \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2\bar{x}_4x_3 \\ = x_1x_4 + x_2x_4 + x_1x_3 + x_2x_3 + x_1x_3x_4 + x_2x_3x_4 + x_1x_2x_3x_4 + x_3\end{aligned}$$

There is no variable which is common to three out of the four doubletons. Thus this function is not M_1M_1R . This exhausts all steps in the algorithm.

CHAPTER 7
CONCLUSIONS

As a result of modern technology in integrated circuits, it is well known that a large amount of logic circuitry can be implemented on a small chip and the cost of a chip does not depend significantly on the number of junctions to be deposited. The module functions M_2 and M_1 are such that they need a large number of transistors and are therefore economical only in integrated circuit form.

It was shown in Ref. 1 that efficient use of M_1 modules results in substantially more economical realizations of combinational circuits. Here it was shown that M_2 is a better class of module functions and normally results in realizations more economical than those using M_1 . It was also shown that for any three-variable switching function one M_2 module is sufficient, and three are sufficient for an arbitrary four-variable function. We did not find a way to use the M_2 module in an optimum manner for arbitrary switching functions.

The properties of total asymmetry and completeness, which are obviously useful for module functions, are responsible for the difficulties in finding good ways to use the modules in the second level. Simple functions like 'OR', 'AND' and 'EX-OR' are symmetric and easily used in the second level. Asymmetric functions, however, do not lend themselves to simple algebraic representations and lead to complicated functional decompositions.

Suggested Areas of Further Research

The following are some of the issues which are yet to be explored:

- a) The use of more than one module kind simultaneously in an efficient manner.
- b) The development of mathematical tools to express a switching function using a totally asymmetric second-level connective.
- c) Efficient use of complemented inputs, or better choice between the two forms of input if only one can be used.
- d) Determination of better module functions.
- e) Module functions which are not defined for three variables, but are only defined for four variables or more.

Out of these and others, the first one seems to be the most important. Since a variety of logic tasks is required in a digital system, the same module function may not be good for all of them. For example, if one has to realize a simple AND, the M_2 module is one of the worst to use. Thus a set of several distinct module functions should offer considerable economy, but the methods for choosing the functions in the set and efficiently using them are not known and are expected to be hard to find.

REFERENCES

1. Patt, Yale N., Minimal Module Synthesis of Switching Functions, Ph.D. Thesis, Stanford University, Dept. of Elect. Eng., 1966.
2. Haring, Donald R., "Multi-Threshold Building Blocks," IEEE Trans. on Electronic Computers, Vol. EC-15, No. 4, Aug. 1966, pp. 662-663.
3. Prather, Ronald E., Introduction to Switching Theory (A Mathematical Approach), Allyn and Bacon, Inc., Boston, 1967.
4. Even, S., I. Kohavi, and A. Paz, "On Minimal Modulo-2 Sum of Products for Switching Functions," IEEE Conference Record of 1966 Seventh Annual Symposium on Switching and Automata Theory, Oct. 1966.
5. Even, Shimon and Albert R. Meyer, "Sequential Boolean Equations," Technical Report RADC-TR-67-629.
6. Cohn, M., Switching Function Canonical Forms Over Integer Fields, Ph.D. Thesis, Harvard University, Dec. 1960. (Theory of Switching Report No. BL-27.)
7. Harrison, Michael A., Introduction to Switching and Automata Theory, McGraw Hill Company, 1965.

APPENDIX A

SOME DECOMPOSITIONS OF SWITCHING FUNCTIONS

R. E. Prather (Ref. 3) has described some decompositions on switching functions which are due to Ashenhurst. All these decompositions are called disjunctive, meaning that in the decomposition of f into functions ϕ_i

$$f(x_1, x_2, x_3, \dots, x_n) = F(\phi_1, \phi_2, \dots, \phi_p)$$

ϕ_1, ϕ_2 through ϕ_p are switching functions of p disjoint subsets of the set of variables such that the subsets form a partition over this set. Disjoint decompositions put a severe restriction on the functions which can be decomposed. What follows is an approach to obtain some non-disjoint decompositions. Also, the method suggested by Ashenhurst involves the use of partition charts and tables, while what follows is somewhat easier to handle.

Notation: Let $f(X)$ denote a function on the set of variables X .

Theorem A.1: A switching function $f(X)$ of n variables can be decomposed as an AND of two functions of $(n-1)$ variables each, i.e.,

$$f(X) = g(X - x_j) \cdot h(X - x_i)$$

if and only if in the expansion over the two variables x_i and x_j not common to g and h

$$f(X) = p + x_i q + x_j r + x_i x_j s$$

it is true that $p \cdot s = q \cdot r$.

Proof: Let f be expanded about x_i and x_j .

$$f = p + x_i q + x_j r + x_i x_j s$$

where x_i (x_j) is the variable which is an argument of g (h) but not of h (g). Then expanding g about x_i

$$g = g_1 + x_i g_2$$

and expanding h about x_j

$$h = h_1 + x_j h_2$$

and $f = g \cdot h$

$$= (g_1 + x_i g_2) \cdot (h_1 + x_j h_2) \quad \text{A.2}$$

$$= g_1 \cdot h_1 + x_j g_1 h_2 + x_i g_2 h_1 + x_i x_j g_2 h_2$$

Since the g's and h's are functions of precisely the same variables as p, q, r, and s, there must be the following one-to-one correspondence between A.1 and A.2, if both must hold for all values of x_i and x_j :

$$p = g_1 \cdot h_1 \quad \text{A.3a}$$

$$q = g_2 \cdot h_1 \quad \text{A.3b}$$

$$r = g_1 \cdot h_2 \quad \text{A.3c}$$

$$s = g_2 \cdot h_2 \quad \text{A.3d}$$

Now the problem of finding the required decomposition reduces to solving equations A.3 for the g's and h's, and the necessary and sufficient conditions for the decomposition to exist are the same as the necessary and sufficient conditions for equations A.3 to have a solution.

Necessity: From equations A.3,

$$p \cdot s = q \cdot r$$

and it is therefore a necessary condition.

Sufficiency: To show that this condition is also sufficient for the equations to have a solution, we show that the following is a solution and exists when the above condition is satisfied:

$$g_1 = p \cup r$$

$$g_2 = q \cup s$$

$$h_1 = p \cup q$$

$$h_2 = r \cup s$$

A.3a $g_1 h_1 = p \cup qr$

But since $qr = ps$,

$$g_1 h_1 = p$$

A.3b $g_2 h_1 = q \cup ps$

But since $ps = qr$

$$g_2 h_1 = q$$

A.3c $g_1 h_2 = r \cup ps = r$

A.3d $g_2 h_2 = s \cup qr = s$

This proves sufficiency of the condition.

The above theorem also suggests a solution for a decomposition which in general is not unique.

Theorem A.2: A switching function $f(X)$ of n variables can be decomposed as the EX-OR of two functions of $(n-1)$ variables each, i.e.,

$$f(X) = g(X - x_j) + h(X - x_i)$$

if and only if $s = 0$, where s is the coefficient of $x_i x_j$ in the IRSE of $f(X)$.

Proof: Let f be expanded about x_i and x_j such that

$$f = p + x_i q + x_j r + x_i x_j s$$

Again g and h when expanded about x_i and x_j , respectively, can be written as

$$g = g_1 + x_i g_2$$

$$h = h_1 + x_j h_2$$

and since $f = g + h$, we have

$$p + qx_i + rx_j + sx_ix_j = g_1 + g_2 \cdot x_i + h_1 + h_2x_j$$

Hence

$$p = g_1 + h_1 \quad \text{A.4a}$$

$$q = g_2 \quad \text{A.4b}$$

$$r = h_2 \quad \text{A.4c}$$

$$\text{and } s = 0 \quad \text{A.4d}$$

Again, the desired decomposition exists if there exists a solution of equations A.4. Equation A.4d demands $s = 0$, and clearly A.4a through A.4c always have solutions, with much choice left in selecting g_1 and h_1 , so that the solution is not unique.

We now consider the decomposition of a n-variable switching function f in the form

$$f(X) = \phi(x_i, x_j, g(X - x_j))$$

The IRSE of ϕ is

$$\begin{aligned} \phi(x_i, x_j, g(X - x_j)) = & a_0 + a_1x_i + a_2x_j + a_3x_ix_j + a_4g + a_5x_ig + a_6x_jg + \\ & a_7x_ix_jg \end{aligned}$$

Let g be expanded about x_i such that

$$g = g_1 + x_ig_2$$

Then

$$\begin{aligned} \phi = & a_0 + a_1x_i + a_2x_j + a_3x_ix_j + a_4g_1 + a_4x_ig_2 + a_5x_ig_1 + a_5x_ig_2 + \\ & a_6x_jg_1 + a_6x_jx_ig_2 + a_7x_ix_jg_1 + a_7x_ix_jg_2 \end{aligned}$$

$$\begin{aligned} \phi = & a_0 + a_4g_1 + (a_1 + a_4g_2 + a_5g_1 + a_5g_2) x_i + (a_2 + a_6g_1) x_j + \\ & (a_3 + a_6g_2 + a_7g_1 + a_7g_2) x_ix_j \end{aligned}$$

Also f can be expanded about x_i, x_j

$$f = p + qs_i + rx_j + sx_i x_j$$

so that

$$p = a_0 + a_4 g_1 \tag{A.5a}$$

$$q = a_1 + a_5 g_1 + (a_4 + a_5) g_2 \tag{A.5b}$$

$$r = a_2 + a_6 g_1 \tag{A.5c}$$

$$s = a_3 + (a_6 + a_7) g_2 + a_7 g_1 \tag{A.5d}$$

Equations A.5a and A.5c demand that either

$$p = r \tag{A.6a}$$

$$\text{or } p = \bar{r} \tag{A.6b}$$

$$\text{or } p = \text{constant} \tag{A.6c}$$

$$\text{or } r = \text{constant} \tag{A.6d}$$

Let us define a relation ' $\overset{*}{=}$ ' such that $p \overset{*}{=} r$ if and only if at least one of the conditions A.6 is satisfied.

Also

$$p + q = a_0 + a_1 + (a_4 + a_5) (g_1 + g_2)$$
$$r + s = a_2 + a_3 + (a_6 + a_7) (g_1 + g_2)$$

These equations also imply a condition

$$p + q \overset{*}{=} r + s \tag{A.7}$$

Therefore A.6 and A.7 are two necessary conditions for the solution, and hence a decomposition, to exist.

To show that these conditions are also sufficient, it should be observed that once these conditions are satisfied, the decomposition can be obtained by solving the equations, though not uniquely. A.5a or A.5c solve directly for g_1 , and A.5b or A.5d can be solved next for g_2 .

Now we consider the special case of four-variable functions. Let it be desired to decompose F as follows:

$$F(x_1, x_2, x_3, x_4) = \phi(x_1, x_2, g(x_1, x_2, x_3, x_4))$$

There always exists a trivial decomposition in this form when $g = F$, but what we are looking for are all the non-trivial decompositions in this form. The function g can be written as

$$g = g_0 + g_1 x_3 + g_2 x_4 + g_3 x_3 x_4$$

where g_0, g_1, g_2, g_3 are functions of the two variables x_1 and x_2 . Then ϕ can be written as

$$\begin{aligned} \phi &= a_0 + a_1 x_1 + a_2 x_2 + a_3 x_1 x_2 + a_4 (g_0 + g_1 x_3 + g_2 x_4 + g_3 x_3 x_4) + \\ &\quad + a_5 x_1 (g_0 + g_1 x_3 + g_2 x_4 + g_3 x_3 x_4) + a_6 x_2 (g_0 + g_1 x_3 + g_2 x_4 + g_3 x_3 x_4) \\ &\quad + a_7 x_1 x_2 (g_0 + g_1 x_3 + g_2 x_4 + g_3 x_3 x_4) \\ &= a_0 + a_1 x_1 + a_2 x_2 + a_3 x_1 x_2 + a_4 g_0 + x_3 (a_4 g_1 + a_5 g_1 x_1 + a_6 g_1 x_2 + \\ &\quad + a_7 g_1 x_1 x_2) + x_4 (a_4 g_2 + a_5 g_2 x_1 + a_6 g_2 x_2 + a_7 g_2 x_1 x_2) + x_3 x_4 (a_4 g_3 + \\ &\quad + a_5 g_3 x_1 + a_6 g_3 x_2 + a_7 g_3 x_1 x_2) \end{aligned}$$

Since the IRSE of F is unique, there exists a one-to-one correspondence between the above terms and those of F when written as

$$F = p + qx_3 + rx_4 + sx_3 x_4$$

Hence

$$p = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_1 x_2 + a_4 g_0 \quad \text{A.7a}$$

$$q = g_1 (a_4 + a_5 x_1 + a_6 x_2 + a_7 x_1 x_2) \quad \text{A.7b}$$

$$r = g_2 (a_4 + a_5 x_1 + a_6 x_2 + a_7 x_1 x_2) \quad \text{A.7c}$$

$$s = g_3 (a_4 + a_5 x_1 + a_6 x_2 + a_7 x_1 x_2) \quad \text{A.7d}$$

The problem of obtaining the decomposition is that of solving these equations for $g_0, g_1, g_2,$ and g_3 .

Obviously there exists a g_0 for every choice of a_0, a_1, a_2, a_3 which satisfies A.7a. These cases are not of interest and are not considered as different decompositions.

Equations A.7b, A.7c, A.7d demand that

$$q \cup r \cup s = (g_1 \cup g_2 \cup g_3) (a_4 + a_5 x_1 + a_6 x_2 + a_7 x_1 x_2)$$

Hence $a_4, a_5, a_6,$ and a_7 must be so assigned that the set of true vertices of $a_4 + a_5 x_1 + a_6 x_2 + a_7 x_1 x_2$ contains the set of true vertices of $q \cup r \cup s$. All such choices for the constants result in solutions of these equations, which then give the desired decomposition. In order that there may exist a non-trivial decomposition, the function $a_4 + a_5 x_1 + a_6 x_2 + a_7 x_1 x_2$ should not be made identically equal to 1, which implies that $q \cup r \cup s \neq 1$ is the necessary and sufficient condition for a non-trivial decomposition of the desired form to exist. This approach can in principle be extended to functions of more than four variables. The choice of constants offers a multiplicity of solutions which increases with number of variables and exhaustion may get out of hand.

SECTION VI

DESIGN AUTOMATION FOR DIGITAL SYSTEMS

William M. Inglis

Design automation is so new and complex a field that it is still more an art than a science. The best techniques are known only in a few cases. In most instances, techniques are used which "seem" good or appropriate. We attempt here to summarize what has been done in the subfield of digital system design automation and to indicate a few techniques for improving the handling of the one part of this subfield which appears to have the weakest existing solutions.

CHAPTER 1

OVERVIEW

A few years ago it became apparent that digital-systems design engineers in industry were spending substantial time doing work which did not call for any special engineering talent, such as the preparation of running lists which specify sets of printed-circuit-board pins which are to be connected by wires. This and other tasks can be defined by reasonably simple algorithms, and therefore can be programmed and executed by a digital computer with only moderate effort. Such programs are referred to in the literature as design automation programs. They accomplish a tedious job more cheaply than when done by hand and free engineers to perform more creative, thought-demanding work.

The purpose of this chapter is to present a brief survey of existing digital-system-design automation programs, point out their strong and weak points, and describe a set of algorithms which could greatly improve the results in one case.

For ease of presentation, we will define seven tasks which comprise most of the jobs done by design automation programs. Methods of performing each task will then be discussed. The seven tasks are:

1. Set up a data structure which specifies the desired digital circuitry and its interconnections. This specification may be:
 - a. directly copied from input cards; or

- b. derived by the computer from more simply specified requirements on the desired circuitry.
- 2. Set up a data structure to describe the available printed circuit boards.
- 3. Assign the designed circuitry to the available printed circuit boards.
- 4. Place the printed circuit boards on a backplane so as to satisfy some goodness criterion, such as
 - a. shortest total wire length
 - b. shortest maximum wire length.
- 5. For each signal set (set of electrically common, but as yet unconnected points on the backplane), find the best order in which to connect the members and produce a tape to control an automatic wiring machine.
- 6. Simulate the operation of the system being designed so that most procedural, logical and component-tolerance errors may be detected before a prototype is ever constructed.
- 7. Prepare adequate quality logic diagrams as documentation for the system being designed.

1.1 THE DATA STRUCTURE DESCRIBING THE DESIGNED SYSTEM

The first job of the design automation program is to build up in the machine memory a description of the digital system being designed. At one extreme, the actual circuitry of the system may be specified completely by design engineers and fed into the computer via cards or tape. At the other extreme, a complex register transfer

language may be used in which the design engineer specifies only the tasks to be done by the system and time and money restrictions on the operation of the system. In this case, the design automation program must derive the precise circuitry to be used from the information supplied.

The first of these cases puts too much of the burden on the engineer (as we wish to automate the design process), while the second may be very costly both to develop and to operate and may strain the computer storage capacities, unless limited in its generality.

The best alternative is probably in the middle: the engineer should always be allowed to specify circuitry precisely, and yet if the design is not critical, the computer should do some of the work for him. An example is provided by the Digital Design Automation Program. Here, all combinatorial logic is implemented with NAND gates, whereas engineers think best in terms of AND-OR logic. The engineer is given the option of specifying the precise NAND network he wishes, or of writing an equivalent AND-OR equation which the computer will convert to NAND. In the latter case, the engineer loses control over the exact network configuration, but gains from the relative ease of specifying the operation to be performed.

So far we have considered how the man-computer team arrives at a completely specified network configuration. Because there are as many data-structure setups to store this information as there are design automation programs, we will limit ourselves to a discussion of the basic features that this data structure should possess in

order to be used efficiently. First we notice that three (at least) of the design automation tasks must refer to the data structure: assignment of circuitry to printed-circuit (PC) boards, simulation of system operation, and preparation of logic diagrams. The chosen data structure should be adequate for use by all three of the corresponding design automation programs. We will now investigate the demands on the data structure made by these three programs.

In order to assign the circuitry described by the data structure to PC boards (which are described by a similar data structure), it must be possible to compare a circuit in the designed system data structure with a circuit in a PC board data structure in order to determine if that particular portion of the designed system can be assigned to the particular PC board. The author does not know of an existing program which accomplishes this in a non-trivial manner and a novel means for achieving this comparison ability is discussed in Chapter 2 of this section. Most existing systems sidestep this issue by requiring the engineer to make manual assignments in many cases. Those assignments which are done automatically do not meet any particular criterion of goodness.

System simulation capability requires that circuit inputs be traceable from circuit outputs in the data structure and that the location of a circuit element in the data structure be easily found when the output name or designator of the circuit element is known. These properties are clearly seen to be necessary if it is to be possible to evaluate the outputs of a circuit when the inputs are known.

Automatic logic diagram preparation can be done in a variety of ways, each placing different requirements on the data structure. Discussion of these requirements will be deferred until the section on automated logic diagram preparation.

1.2 THE DATA STRUCTURE DESCRIBING AVAILABLE PRINTED CIRCUIT BOARDS

The PC board data structure is identical in basic form to the designed system data structure. In the former case, however, circuit inputs and outputs are represented by board pin numbers, while in the latter case signal names are used.

Normally, a library of standard PC board data structures is maintained on tape. Design engineers then have the option at run time of declaring any standard board to be unavailable for a particular job and of inputting the descriptions of special boards which may be needed but are not presently on the tape.

Depending on the sophistication of the programs, PC board descriptions may contain such information as the locations of test points, the presence or absence of pull-up resistors on gates, cost factors to be used in determining when to use the PC board, etc.

1.3 ASSIGNMENT OF DESIGNED CIRCUITRY TO AVAILABLE CIRCUIT BOARDS

A wide variety of algorithms can be used to perform the assignment function. In the very simplest program, the engineer is required to associate every element in his designed system with an element on a printed circuit board. In that case, the only design automation achieved is automatic linking of PC board leads with element leads of the designed system.

From here, the first step upwards in complexity is achieved by considering the modularity of PC board subnetworks. That is, each PC board can be thought of as containing n independent digital circuits, each of which may be used independently of the others.

We may ease the job of the engineer by requiring him to assign only one pin of each PC board subnetwork to a node of the designed system. For example, in assigning the PC board subnetwork of Figure 1.1A to the designed system of Figure 1.1B, the engineer merely specifies the correspondence: $A \leftrightarrow 1$, i.e., pin 1 is assigned to node A of the system. The design automation program then examines the data structures describing the PC board and the system, and deduces from the correspondence $A \leftrightarrow 1$ that the following correspondences must also exist = $F \leftrightarrow 6$, $G \leftrightarrow 7$, $H \leftrightarrow 8$, $I \leftrightarrow 9$, $J \leftrightarrow 10$, $K \leftrightarrow 11$, $E \leftrightarrow 12$. The engineer is thus saved the tedium of writing down most board-pin to system-node correspondences. The design automation program can determine the remaining correspondences with reasonable efficiency by an exhaustive trial and error method as long as the PC board subnetworks are small.

More sophisticated methods are difficult to devise when arbitrary feedback and fanout of signals are allowed and when there are elements (such as flip-flops) with distinguishable input or output lines.

To move one step higher in complexity, we now require no manual inputs from the design engineer at all. In performing the assignment task, there are two levels of strategy. At the outer level there

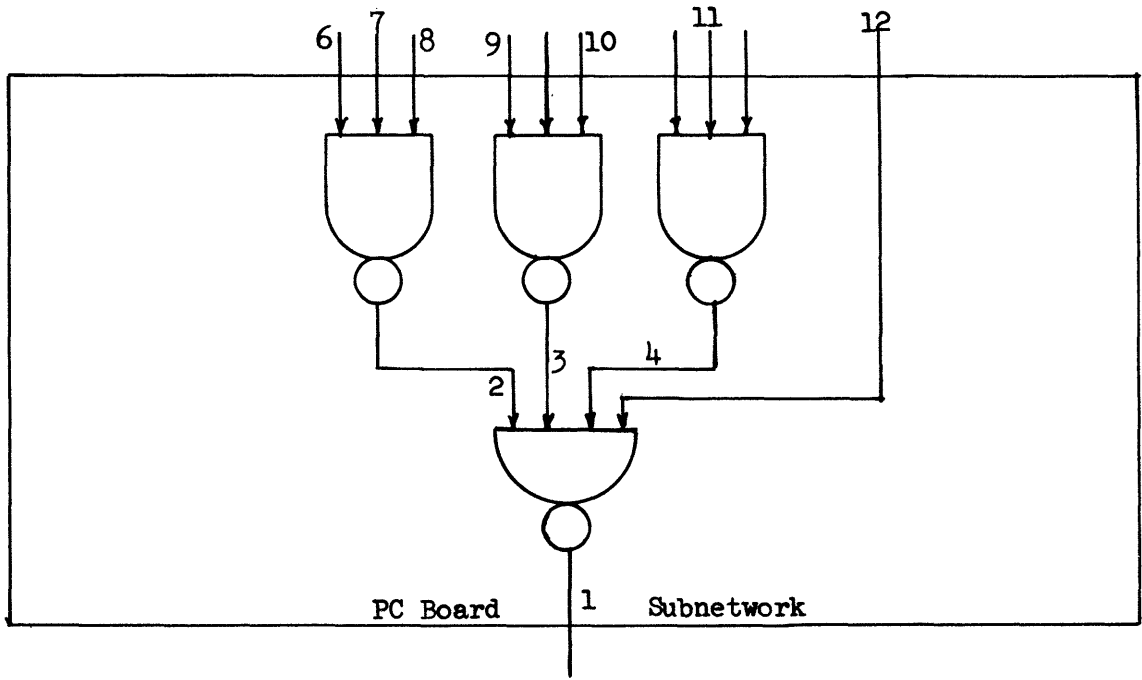


Figure 1.1A

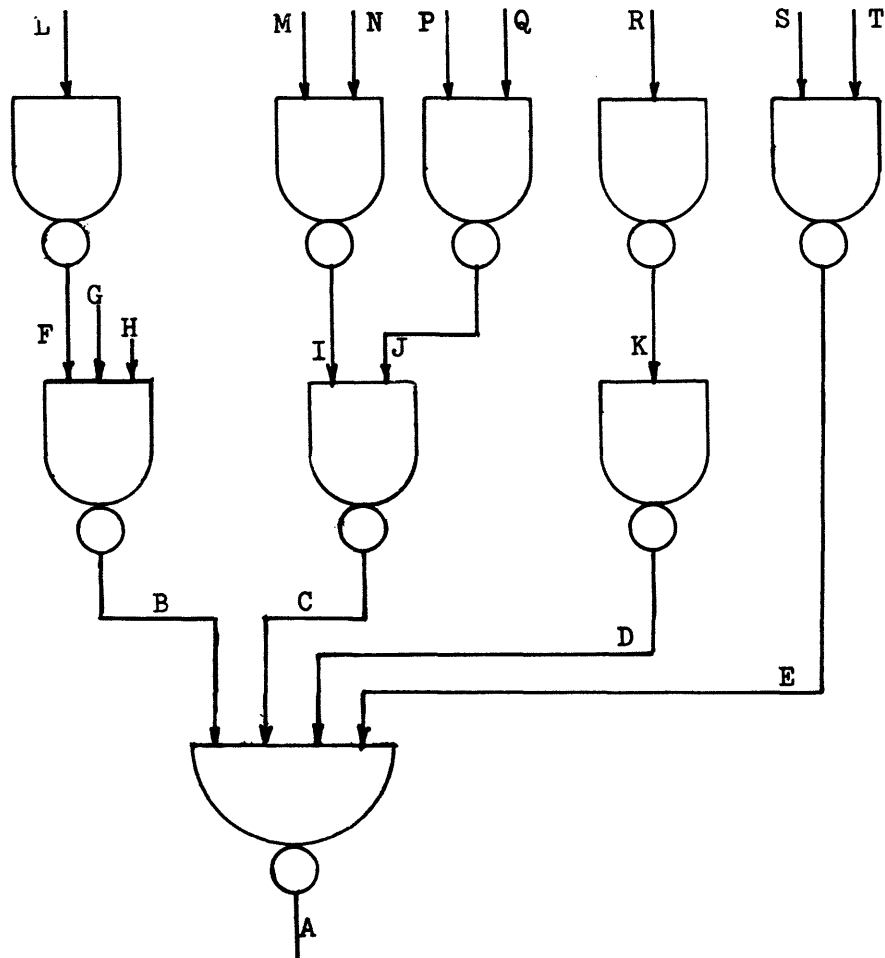


Figure 1.1B

must be rules to determine in what order we will attempt to assign PC boards to the system. As an example, consider the following strategy devised to implement the designed system as cheaply as possible.

First we divide the cost of each circuit board by the number of elements on the board to obtain the cost per element, c_e , of the board. We then determine the fraction f_i of elements of type i in the designed system. Let n_i be the number of elements of type i on the board. Let the quantity $c_e / \sum f_i n_i$ with the sum over all i be a measure of cost for each circuit board. (Note that if f_i is zero for all nonzero n_i , the cost factor of the board will be infinite; that is, the board will never be used.) We arrange the boards in ascending order according to this cost measure and use an assignment algorithm to assign the first board on the list to the designed system as many times as possible. When it is no longer possible to assign the first board, move on to the second one, etc., until all elements are assigned. If any elements are left over, the process can be repeated, this time without demanding that all the subnetworks on a board be used. This process should give a nearly optimal assignment.

The inner level strategy is the strategy used to assign a particular PC board subnetwork to the system. (The outer level strategy determines which particular PC board subnetwork will be assigned at any time.) The inner level strategy is required to decide whether or not a subnetwork will be assigned to the system once it has been determined that it can be assigned.

For example, a NAND gate with ten inputs could be assigned to a two input NAND function in the system but it would most likely be too uneconomical to do so. The inner level strategy must be able to handle all such decisions.

In Chapter 2 a different approach to the assignment problem is taken, which avoids the need for strategies, by considering all possible assignments and selecting the best.

1.4 PLACEMENT OF PC BOARDS ON A BACKPLANE

At this point one might raise the objection that if optimality is to be achieved in the performance of the assignment and placement tasks, they must be considered together and not dealt with independently. This is a valid but unrealistic objection. The number of ways of placing 100 PC boards in 100 slots is $100!$. If we multiply this number by the number of ways of assigning the designed system to available PC boards, the number of combinations becomes unmanageable. No algorithm has yet been devised to handle a significant fraction of this number of combinations in such a manner that optimality is approached in a reasonably short time. Completely acceptable assignment-placement combinations are obtained by dividing the problem into parts. Although absolute optimality will not in general be obtained, it is possible to come close enough so that expenditure of additional funds to improve the solution would not ordinarily be warranted.

The goal of the placement program is to assign the PC boards which implement the designed system to a set of card slots which lie in an x-y plane, each slot having coordinates specified by the design

engineer. The criterion of goodness is usually taken to be the total length of wire necessary to interconnect the PC cards.

There are many algorithms available for obtaining nearly minimum wire length. An example follows. First the engineer is permitted to pre-place as many boards as he wishes. He may, for instance, wish to guarantee that two boards which could cause crosstalk problems are not placed near each other. If no cards are pre-placed, the algorithm selects the two cards having the largest number of electrically common pins and places them at the center of the plane. After these cards are placed (or if there are already some pre-placed cards), the algorithm selects the unplaced board which has the most points in common with the set of placed boards and places it so as to minimize the total length of wire needed to connect it to the previously placed boards. This procedure then continues until all boards have been assigned.

The above algorithm can be improved by adding tie-breaking procedures. Whenever two or more cards each have maximum connectivity with the placed cards, the algorithm assumes that each one in turn is placed and investigates the next step. The best next step then determines what will be done at the present step. If there are ties for the best next step, the tie-breaking procedure can be used again, etc., and for as many levels as computer time **allows**.

1.5 DETERMINATION OF OPTIMUM WIRE ROUTING

Ideally, whatever algorithm is used to accomplish this task should also be used in the previous task whenever it is necessary to calculate wire lengths. For systems of substantial size, however,

this would require much more computer time than the improved results would justify. Results which are nearly as good can be obtained by performing the placement task first and then the wire routing.

The following is a simple wire routing algorithm. The two points closest to each other are chosen for the first wire. The succeeding wires are then placed between a new point and either end of the existing chain such that the new wire is as short as possible. This method may be improved by adding a look-ahead feature. For example, two wires could be placed at a time, selected such that the sum of their lengths is a minimum. If all sets of common points contain a small number of members, an exhaustive or near-exhaustive technique could be employed.

The above algorithm assumes that at most two wires can be attached at any point on the backplane. If this is not in fact a restriction in a particular case, other algorithms can be used and improved solutions obtained.

1.6 SIMULATION OF THE DESIGNED SYSTEM

A digital system simulator will accept input waveforms and initial system conditions and calculate and print out the corresponding output waveforms. Accuracy of the system design can then be checked without having to breadboard the system.

To use such a system simulator, an engineer will specify:

- a. The portion or portions of the designed system he wishes to simulate.
- b. The number of time units to be simulated.

- c. Bivalued waveforms representing the inputs to the system.
- d. Element parameters (e.g., input-output delay of gates, pulse time of one-shots, etc.)
- e. A list of the elements whose outputs are desired.

The simulator will use the designed system data structure to update the output of each element at each time unit.

If the initial output of each element is specified, there will be no problem in finding the proper order in which to update element outputs, for any order will do. The output of each element at time n is then a function of the inputs at time $n-1$.

The more sophisticated simulators have standard error flags to call the attention of the engineer to possible design errors of a procedural nature; for example, trying to exceed the duty cycle of a one-shot, supplying too short a triggering pulse to a one-shot or flip-flop, or triggering both inputs of an R-S flip-flop at the same time.

Design errors of a logical nature are detected by comparing the desired output waveforms to those actually calculated by the simulator. Discrepancies may reveal missing or incorrectly placed elements.

It is also possible to use a simulator to test a designed system for errors due to component tolerances. For this, it is necessary to use parametric flip-flop, gate, and one-shot models. That is, the engineer must be able to change the characteristics of an element at least slightly by specifying parameters for the element. To test for component tolerance errors, the engineer specifies a minimum and a

maximum value for each element parameter. For each element in the designed system, the simulator then chooses the minimum or maximum value of each parameter at random, the simulation is carried out, and the response is retained. Then parameters are again chosen at random for each element and the system simulated again. Discrepancies in the two sets of responses are noted because they could possibly be due to poor design. The more times this procedure is repeated, the better. The number of iterations, of course, depends on the amount of computer time available.

There are two extreme philosophies involved in computer simulation of digital systems. The first maintains that speed of simulation is of utmost importance. Simulators written with this in mind have very simple, approximate element models and all is sacrificed for speed. The second philosophy maintains that accuracy of simulation is essential. Simulators written on this basis have very complicated element models which very nearly approach the actual circuits. The time required to simulate a system is, however, much greater with this second approach.

The Digital Design Automation System employs two simulators, one of each type. The fast simulator is used for very, very large systems which could not be simulated at all with a slower system. The accurate simulator is used for small systems, for critical portions of larger systems, and for more accurate simulation of those portions of very, very large systems which the fast simulator indicates are faulty.

1.7 AUTOMATIC PREPARATION OF LOGIC DIAGRAMS

Many schemes have been employed in this area, most of them differing in what they accept as inputs. One program requires the engineer to locate elements on a standard grid. It then uses the data structure to determine how to draw in the connecting lines, employing one of the existing maze-tracing algorithms. Logic diagrams drawn by such programs appear rather unnatural and artificial due to the random paths followed by the lines connecting elements.

A higher degree of automation is attained when the program assumes the burden of arranging the digital element symbols on the page as well as drawing in the connecting lines. It is usually necessary for the engineer to provide a list of elements all of which are to be placed on a given page. Otherwise a logic page may contain an inconvenient variety of logic units together, such as a few memory circuits, the thirteenth adder bit, and the control logic for the divide instruction. The author has devised a rather complicated algorithm to obtain a reasonable arrangement of specified logic elements on a page, but the algorithm results in a rather low density of elements per unit area on the page as compared to the efforts of draftsmen.

As an example we will now describe an algorithm which will draw simple gating networks in such a manner that no connecting line crosses any other connecting line. A simple gating network is defined to be a network of gates without feedback or fanout. A gate is defined to be at level $n+1$ if its output takes n gate delays to propagate to the final output of the network. The first gate drawn is placed at the

lower right hand corner of the page. All gates are drawn with inputs entering and outputs leaving vertically downward. A variable called LEFT (in units of the width of a gate symbol) is maintained and indicates the horizontal distance from the right edge of the page at which a gate is to be drawn. Similarly, a variable VERT is maintained indicating the distance from the bottom of the page (in units of the height of a gate symbol) at which a gate is to be drawn.

The algorithm is given in the flowchart of Figure 1.2. An example of a gating network drawn with this algorithm is shown in Figure 1.3. The numbers in the figure indicate the order in which gate symbols and connecting lines are drawn.

Similar but more complicated algorithms handle diagrams involving fanout and feedback as well as other element types such as delay lines, one-shots and flip-flops.

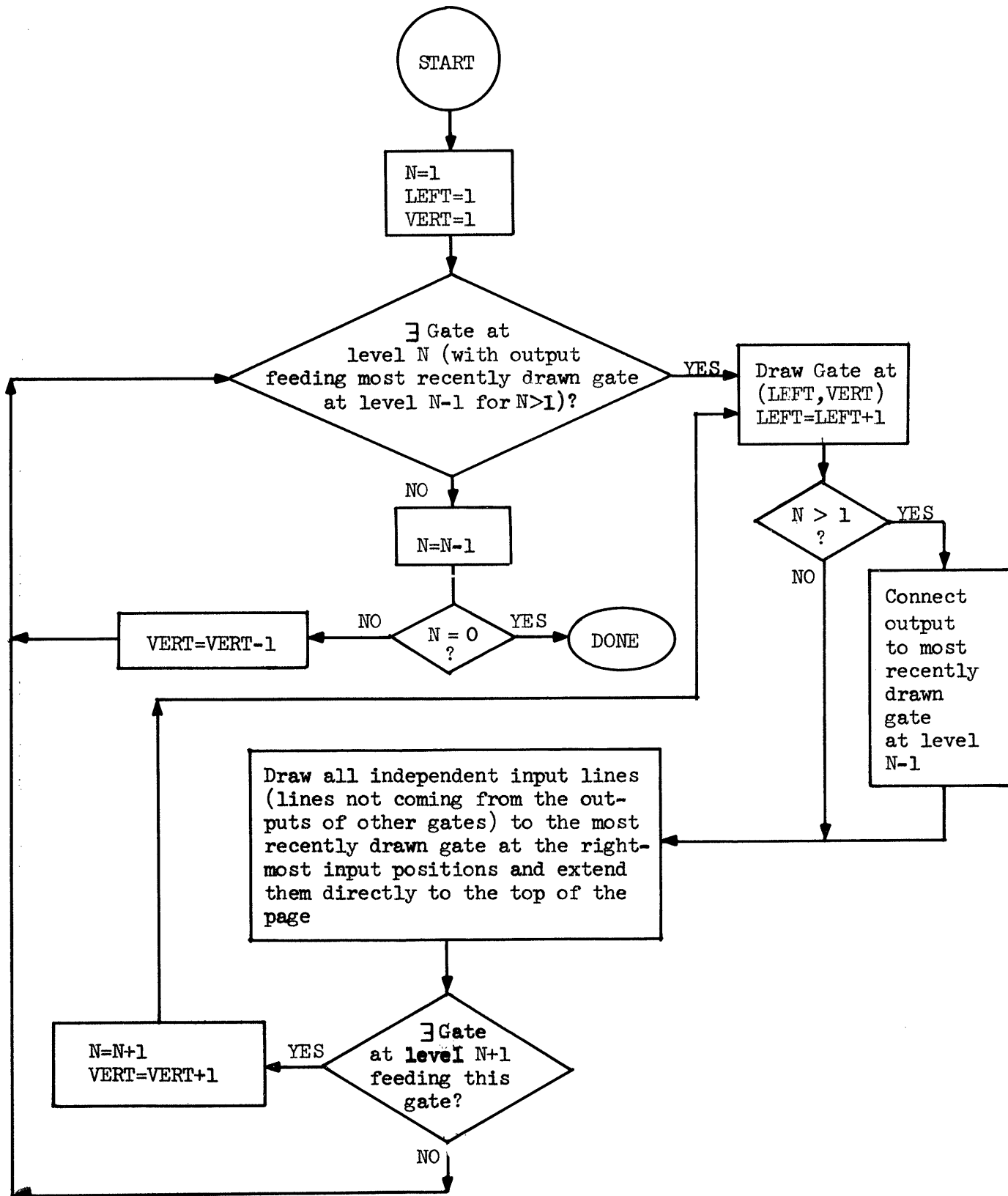


Figure 1.2 Flowchart for Gate Network Algorithm

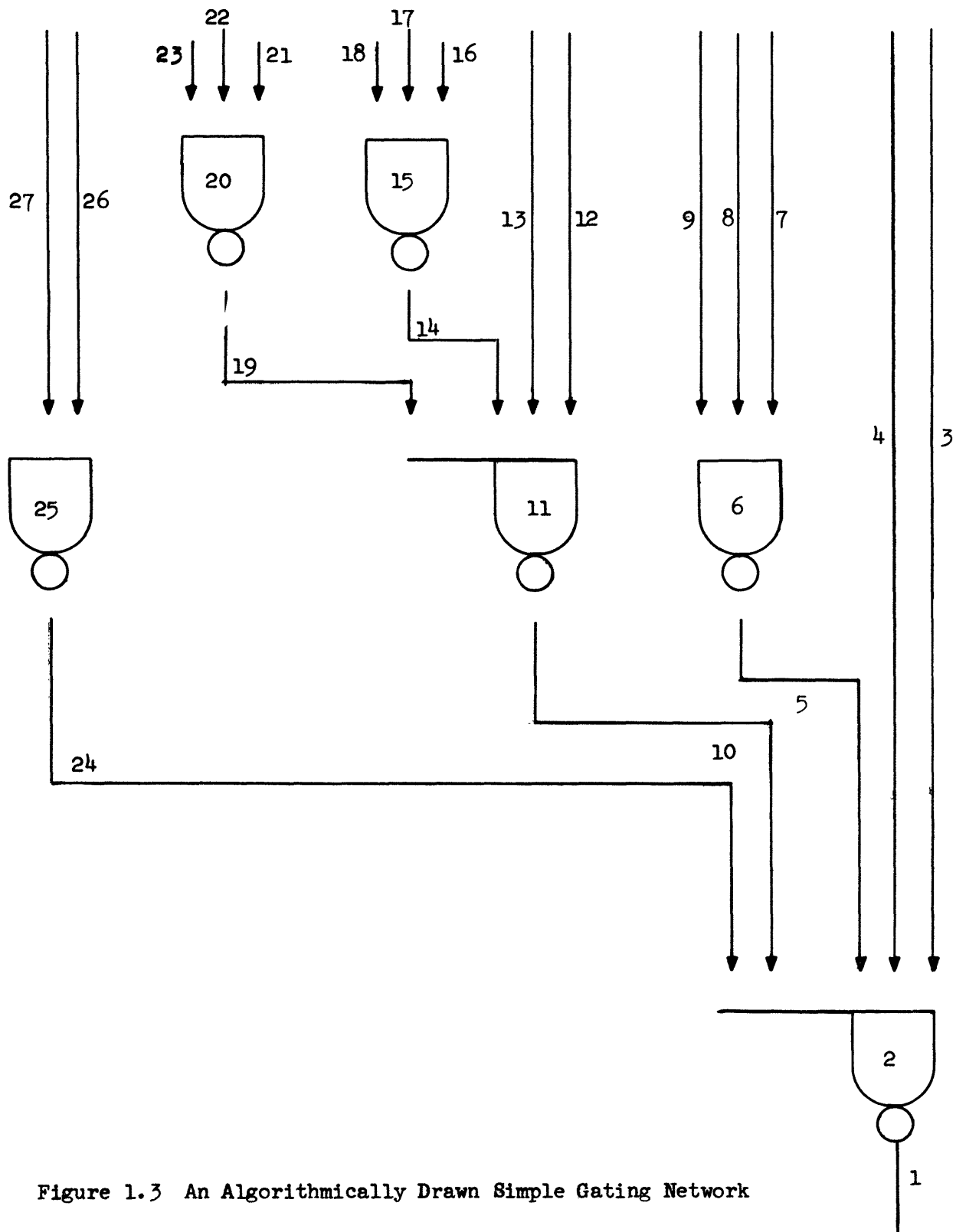


Figure 1.3 An Algorithmically Drawn Simple Gating Network

CHAPTER 2

AN ASSIGNMENT ALGORITHM BASED ON LINEAR PROGRAMMING

In this part of the paper we will point out specific techniques by which a design automation program can be greatly improved. The assignment problem (of the designed system to the available circuit boards) was chosen for inclusion here because the author knows of no existing implementation which assigns in an optimal or near optimal manner and does all assignments automatically.

The discussion will consist of three basic parts:

1. The description of a data structure well suited for comparing designed system networks to printed circuit board networks.
2. Methods of carrying out the comparison above.
3. A description of the assignment algorithm and how it achieves an optimal assignment.

2.1 A COMPARISON-ORIENTED DATA STRUCTURE

Suppose that two AND-OR expressions are given and it is desired to determine if both expressions describe the same gating network. For example $(a+b)(c+d+e)$ and $(f+g+h)(j+k)$ are clearly equivalent expressions. But the issue is confused by the fact that the AND operation is commutative, so that the order of the terms is immaterial. This prevents a straight character-by-character comparison of the two expressions. The two AND terms in one of the expressions would have to be interchanged if such a comparison were to work. What we need is a canonical form: a set of rules for writing logic equations which

guarantee that terms separated by commutative operators will always appear in the same order no matter who writes the expression.

To obtain the canonical form, we first give rules for writing logical equations in a modified reverse Polish notation:

1. Write the number of inputs to the output element of the gating network followed by a letter indicating the logical operation performed by the output element. This is the operator for the Polish form. The number part of the operator tells how many operands follow.
2. The operands are simply a list of what the inputs to the gate are. If an input is just a simple variable, the name of the variable is written. If an input is the output of a gating network, we write it down using rule 1 recursively.

As an example, the expression $(b+c)(d+e)+f$ would become

$$2\phi-2A-2\phi-b-c-2\phi-d-e-f$$

where ϕ indicates OR and A indicates AND. This expression is more understandable if parentheses and commas are used:

$$2\phi[2A\{2\phi(b,c),2\phi(d,e)\},f]$$

The arguments of each operator are the expressions within the following brackets separated by commas. Notice that ordering ambiguity is still present. The expression

$$2\phi-f-2A-2\phi-d-e-2\phi-b-c$$

is also a valid expression for the given equation.

The expression can be easily converted to a canonic form, however. First we define the list of operands following an operator to

be the field of that operator. To put an AND-OR expression in canonical form:

1. Convert the expression to modified reverse Polish form.
2. For each operator in the resulting expression, order the elements in its field according to the following rules:
 - a. all independent variable names come first (are leftmost).
 - b. all A operators lie to the left of all \emptyset operators.
 - c. If P is an operator (A or \emptyset) and $k < m$ are integers, then kP goes to the left of mP .

These rules leave ambiguous the ordering of identical operators appearing in the field of a given operator. Ambiguity is removed beginning at the deepest reduction level and working outward. If in any set of ambiguous operators there are one or more operators with ambiguities in their fields, that set of ambiguous operators cannot be resolved. Since all expressions of interest are finite, however, we can proceed in such a manner that all ambiguity can be removed; namely, beginning at the deepest reduction level and working outward.

Assume now we have two identical operators in the field of some other operator and no ambiguity in the field of either.

We then have two expressions of the form:

- a. $kP-E_1-E_2-E_3-E_4-E_5-E_6-\dots-E_n$
- b. $kP-F_1-F_2-F_3-F_4-F_5-F_6-\dots-F_n$

where the E's are variable names or operators and $E_1-\dots E_n$ represents the canonical ordering for the field of kP . If E_i happens to be an

operator in the above expression then E_{i+1} is in the field of E_i ; we are including all those elements even indirectly in the field of kP .

To decide if a) goes to the left of b) or vice versa we compare E_1 and F_1 ; if one of these goes to the left of the other by rules 2a, 2b, or 2c above, then the expression in which it appears goes to the left of the other. If E_1 and F_1 are identical, we compare E_2 and F_2 , etc. If all elements E_i and F_i are identical, then a) and b) represent identical subnetworks and their order is thus immaterial.

Example:

- 1.) 5A-A-B-C-2 ϕ -C-D-2 ϕ -E-F
2.) 5A-G-H-J-2 ϕ -K-L-3 ϕ -M-N-P
- ↓
↑

Here, expression 1 goes to the left since 2 ϕ goes to the left of 3 ϕ .

Two such expressions can, of course, be of different length. If order has not been resolved by the time we reach the end of the shorter expression, then the shorter one goes to the left.

Note that the foregoing can be easily generalized to cases in which more than two types of gates are used.

2.2 COMPARISON OF CANONICAL GATING EXPRESSIONS

We now have a way of writing combinatorial expressions so that a simple character-by-character comparison will decide whether or not any two expressions represent the same gating network. This does not entirely solve the comparison problem. In practical situations things are much more complicated.

1. There are elements with distinguishable inputs.

2. The logic contains feedback and fanout which has not been considered here.
3. We may wish to compare a PC board subnetwork expression containing, say, two gates and another containing three gates to a designed system expression containing five gates.

Solutions to the above complications exist but they are messy and theoretically uninteresting, so they will not be discussed further here. We will now move on to a discussion of the optimizing assignment algorithm assuming complete comparison capabilities.

2.3 THE OPTIMIZING ASSIGNMENT ALGORITHM

The basic approach taken here will be to set up a linear program with the expression to be minimized representing the cost of the designed system and the constraint equations representing all possible assignments of the designed system to the available circuit boards.

We first define the variable x_{ijk} to be 1 if the j th subnetwork on circuit board k is assigned to node i of the system being designed and 0 if not. By assigning a subnetwork to a node of the system we mean that:

1. The subnetwork has a uniquely defined output pin.
2. The subnetwork can be assigned to the designed system in such a manner that the output pin corresponds to the node in question.

Furthermore x_{ijk} exists as a variable of the linear program if and only if the above assignment is possible.

It is then clear that for each node of the designed system we must have

$$\sum_j \sum_k x_{ijk} = 1$$

The summation is over those j and k for which x_{ijk} exists.

Assigning the jth subnetwork on board k to node i in general implies that some other system nodes will be covered (i.e., will have some circuitry assigned to them indirectly). For instance, in Figure 2.1, if subnetwork N1 of board 1 is assigned to node A of the system, node B will be covered.

Clearly a node can have nothing assigned to it if it is covered and vice versa. Thus the constraint equations are modified from the above:

$$\sum_j \sum_k x_{ijk} + \sum_p \left(\sum_j \sum_k x_{pjk} \right) = 1$$

The second summation is over those nodes p for which subnetwork j of board k will cover node i when assigned to node p.

To set up the constraint equations for the linear program, we use the comparison algorithm to attempt to assign every subnetwork on every circuit board to every system node thus determining which of the x_{ijk} exist.

The cost expression to be minimized is complicated by the fact that whenever a subnetwork j from board k is used, an entire copy of board k is effectively used regardless of whether its other subnetworks are physically attached to the system.

Let us then define

$$x_k = \max_j \sum_i x_{ijk}$$

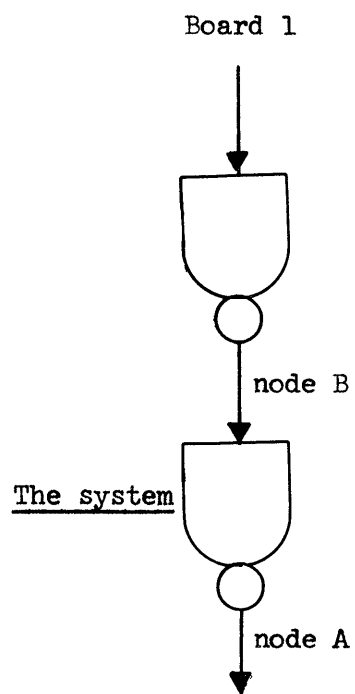
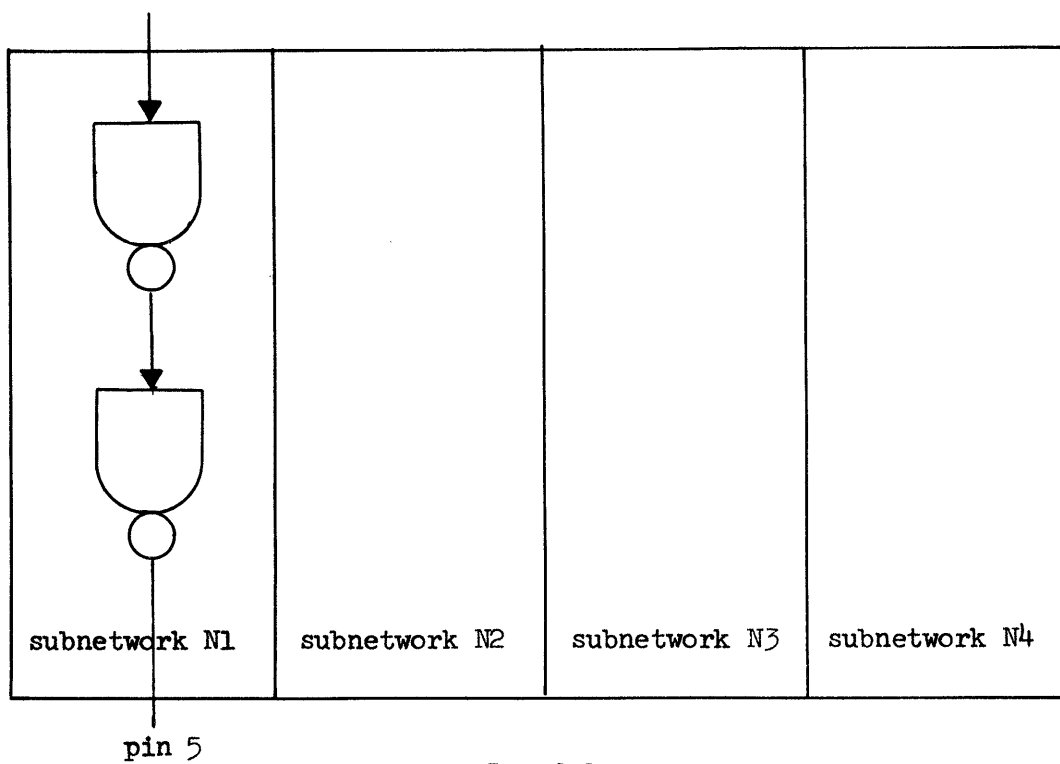


Figure 2.1 An example of a "covered" node.

In other words the number of type k circuit boards used, x_k , is equal to the maximum required number of any one of its subnetworks. For instance, if board 1 has subnetworks A, B, and C and we have assigned three copies of A, three copies of B, and four copies of C, we need four boards of type 1.

Since we are setting up a linear program, we must specify x_k in a slightly different (linear) manner. Namely, we add the following constraint equations:

$$\begin{aligned}x_k &\geq \sum_i x_{i1k} \\x_k &\geq \sum_i x_{i2k} \\&\vdots \\x_k &\geq \sum_i x_{ij_{\max}k}\end{aligned}$$

There is one such set of expressions for each k, that is, for each type of circuit board.

In minimizing the cost function, the linear programming mechanism will automatically select the only consistent value for x_k , namely

$$\max_j \sum_i x_{ijk}.$$

The cost function may now be easily written as:

$$F = \sum_k c_k x_k$$

where c_k is the cost of a single copy of circuit board k.

If all the comparisons are done exactly and all the x_{ijk} 's determined correctly, the Simplex method will give the cheapest

assignment. It is highly likely, however, that any practical problem of this nature would be so complex as to require an inordinate amount of computer time to solve exactly. It is possible to obtain an acceptable solution and begin converging towards the best solution by carrying out only a portion of the Simplex iterations in much less time, however. It is probable that the improvements from iteration to iteration will become smaller and smaller. A cutoff point can then be defined based on the percentage improvement obtained from the last n iterations, and a near-optimal assignment obtained in a reasonable amount of time.