AN ABSTRACT OF THE THESIS OF

Sung Chiao Hu (Name) for the Doctor of Philosophy (Degree)

in Electrical and Electronics Engineering (Major) presented on May 1, 1970 (Date)

Title: CELLULAR SYNTHESIS OF SYNCHRONOUS SEQUENTIAL MACHINES

Abstract approved: Redacted for Privacy

Robert A. Short

With the advancing solid-state technology, it is necessary to develop new techniques for synthesizing digital networks. The regular pattern of cellular circuits seems to be the best fitted for the new LSI technology. Recently, cellular implementations of combinational circuits have received considerable attention but very little attention has been given to sequential circuits. In this paper, we present two new methods for realizing sequential machines, both using cellular circuits. These new techniques will also enable us to do away with the time-consuming and difficult problem of state assignment. State-assigned (Moore) machines are assumed throughout.

The first method converts sequential functions into combinational like equations. In order to do so, the machine must be either definite or finite input and feedback memory (FIFM). If the machine is neither definite nor FIFM, it is made FIFM by constructing a proper feedback

function. These combinational like equations can easily be implemented by conventional combinational cellular circuits, such as the cutpoint cellular arrays, together with delay elements.

The second method utilizes matrix methods. It is noted that when a machine is in a certain state and is subject to an input, it does two things: it makes a state transition and it produces outputs. If the diagonal elements of an nxn array of cells are thought as representing n states, the transition of states can be accomplished by first moving horizontally and then vertically and the output can be collected by an added bottom collection row.

In both cases, bounds on the number of cells are established and minimal realizations are studied. Methods for starting these cellular machines are also investigated. In order to make the machine more flexible, techniques are devised to initialize the machine into any state desired.

It is safe to predict that future computing systems will continue to increase the demands on several sophisticated design areas. They will need to be more readily expandable and modifiable. Automatic error detection and correction will also play a more significant role. Therefore, besides modularity, reliability and programmability are also important aspects of any new design techniques. Both synthesis methods presented in this paper can easily be modified to include these features.

Cellular Synthesis of Synchronous Sequential Machines

by

Sung Chiao Hu

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

June 1970

APPROVED:

Redacted for Privacy

Professor of Electrical and Electronics Engineering

in charge of major

Redacted for Privacy

Head of Department of Electrical and Electronics Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented May 1, 1970

Typed by Barbara Eby for Sung Chiao Hu

ACKNOWLEDGEMENTS

The author acknowledges with sincere thanks the interest and guidance offered by Dr. R. A. Short; the willingness of Drs. F. Young, J. Herzog, D. Pierce and M. Cropsey to serve on the author's graduate committee; the help offered by Mr. W. Chiu in the final stages of the thesis preparation; and the patience and understanding of my loving wife, Lily.

TABLE OF CONTENTS

LIST OF FIGURES

CELLULAR SYNTHESIS OF SYNCHRONOUS SEQUENTIAL MACHINES

I. INTRODUCTION

1.1 Problems, Objectives, and Results

The two major problems in designing a synchronous sequential machine are state reduction and state assignment. The latter is especially difficult and is still a subject of intensive research. However, if we can devise a new synthesis technique which is independent of the assignment of states, then we do not have to bother with this difficult state assignment problem.

It cannot be denied that the current computer hardware technology is going toward large scale integration (LSI). The advantages of LSI are many and are well known. Cellular circuits seem to be the best fitted for this new technology. But, though work has been done in cellular combinational circuits, there have been few results, as yet, in the cellular synthesis of sequential machines.

It is safe to predict that future computing systems will continue to increase the demands on several sophisticated design areas. They will need to be more readily expandable and modifiable. Automatic error detection and correction will also play a more significant role. Therefore, modularity, reliability, and programmability are all important aspects of any new design techniques.

Our objectives are to derive new synthesis techniques which do not require state assignment and can be achieved by interconnecting uniform building blocks in a regular pattern. For easy addition or reduction of building blocks as desired, we emphasize the uniformity and the regularity requirements. For simplicity, we consider only binary, finite-state, deterministic synchronous sequential machines of the Moore type. All of our techniques, however, can be extended to include other types of machines.

Two synthesis methods are developed, each is able to achieve the above objectives. The first method converts sequential functions into combinational-like equations. These combinational-like equations can then be implemented by conventional combinational cellular circuits together with delay elements. The second method employs an array of cells in which state transitions are accomplished by "row operations" and outputs are obtained by "column operations". Both methods are readily expandable to include features such as reliability and programmability. In each case, bounds on the number of cells are established and minimal realizations are studied. Methods of starting these cellular machines from a given state are also investigated.

For easy reference, all Figures are collectively given at the end of the text material and all example machines are listed in Appendix I. Except where specific references are made, all developments presented in this paper are original.

1.2 Definition of Terms

We define the following basic terms that are important to the understanding of this paper.

A sequential machine, SM, is a quintuple $M=(I, Z, S, f, g)$, where

$I = (i_1, i_2, \cdots, i_p)$ is a finite set of input symbols,

$Z = (z_1, z_2, \cdots, z_q)$ is a finite set of output symbols,

$S = (s_1, s_2, \cdots)$ is a set of states,

$f : SXI \rightarrow S$ is the next-state function or the transition function, and

$g : S \rightarrow Z$ is the output function.

A sequential machine in which the state set S contains only a finite number of elements is called a finite-state machine. A machine that satisfies all of the above requirements is called a Moore machine. If f is defined for all values of SXI the machine is transition complete; otherwise it is transition incomplete and the undefined values are called "don't cares". Similarly, if g is defined for all values of S, the machine is output complete; otherwise it is output incomplete and the unspecified values are also don't cares. If both transition and output are complete, the machine is completely specified; otherwise it is incompletely specified. Since the transition function and the output function characterize a machine, they are referred to as characterizing functions. A machine whose characterizing functions are not

subject to any uncertainty is deterministic. When inputs and outputs of a machine are restricted to binary signals, the machine is a binary machine. A sequential machine is synchronous if the transition of states occurs only at prescribed instances of time controlled by a clock. The time unit in synchronous sequential machines is one clock time (see Figure 1).

A sequential machine is often described by a flow table. A flow table for a Moore machine is the tabular form of Figure 2. Columns labelled by input symbols are next-state columns; columns labelled by output symbols are outout columns; each row represents a state of the machine; each entry at the intersection of the h-th input column and the j-th state row is the next-state $f(s_j, i_h)$; the entries in the output columns at the j-th state row are outputs $g(s_j)$.

A next-state column is a permutation column if every state of the machine appears exactly once in that column; a next-state column is a reset column if all entries in that column are the same. A machine that consists only of permutation columns is a permutation machine; a machine that consists only of reset columns is a reset machine; a machine that consists only of permutation and reset columns is a permutation-reset machine or simply a P-R machine. Zeiger in 1967 [29] showed that any machine can be composed of cascaded connections of only P-R machines.

A machine can also be represented by a state graph, which is an alternative representation of the flow table. An example of a state graph for a Moore machine (Figure 3a) is shown in Figure 3b. Each node in the graph is a state-output pair. There are as many nodes as there are states. There is an arrow labeled by i_h going from node s_j to node s_k if and only if $s_k = f(s_j, i_h)$. If there exists a finite sequence of arrows going from node s_u to node s_v, state s_u is said to be a predecessor state of the state s_v, and state s_v a successor state of the state s_u. If there exists such a sequence in which the number of arrows is exactly one, then state s_u is an immediate predecessor of the state s_v and state s_v an immediate successor of the state s_u. A state s_j is called a transient state if there is a state $s_i \in S$ such that s_i is not a predecessor state of s_j.

In circuit realizations of machines, sometimes an output is routed back into the circuit and is used as a decision influence parameter. This is called feedback. The function that describes this feedback is the feedback function. Since a feedback is essentially another input to the circuit, it is often referred to as a feedback input. If a machine can be realized without any feedback inputs, it is said to be feedback free. Friedman in 1966 [7] showed that any finite-state machine can be realized by sequential circuits with at most one feedback input. If none of the available output functions can be used as the feedback function, then an additional function must be constructed. If

we let F be this feedback function, then a machine is finite input and feedback memory (FIFM) if and only if its characterizing functions can be described by the present inputs and a finite number of past inputs and a finite number of past feedbacks. That is, for every non-transient state s at any given time t,

$$s_t = G(I_t, I_{t-1}, \cdots I_{t-k_1}, F_{t-1}, F_{t-2}, \cdots, F_{t-k_2}),$$

where k_1 and k_2 are smallest integers such that the above equation holds for all non-transient states of the machine. k_1 is called the input memory of the machine; k_2 is the feedback memory of the machine. Let

$$K = \max(k_1, k_2),$$

K is the memory of the machine. When K=0, the machine is combinational; otherwise it is sequential. If $k_2 = 0$, it is feedback free and is often called definite. Thus, in a feedback free machine, all non-transient states can be distinguished by applying a finite number of successive inputs. Transient states can be entered only by external manipulations and will be discussed later.

1.3 Historical Background

Aside from the general switching theory, the following areas are of special interest to this study.

A. Machine Decomposition. The decomposition of machines by partition was first introduced by Hartmanis and later expanded by Hartmanis and Stearns [10]. Gill [8], Yoeli [27] and Kohavi and Smith [13] also made considerable contributions in this area. Another approach to the problem of decomposition of machines is by the theory of semigroups. Krohn and Rhodes in 1962 [14] proved an abstract theorem which in effect says that any machine can be composed of cascaded connections of only two types of machines. This theorem was also proved by Zeiger in 1967 [29] who introduced P-R machines. Most of the work in this area are well discussed in the book by Hartmanis and Stearns [10]. The theory of machine decomposition provides us a good understanding of machine structures and difficulties involved in cellularizing sequential circuits by way of decompositions.

B. Definite Machines. A pioneer paper in the area of feedback free machine was given by J. M. Simon (1959) in his "A Note on the Memory Aspects of Sequential Transducers" [23]. Gill treated this subject in his 1962 book and called it an "output-independent machine" [9]. A more complete but abstract treatment of the topic was given by Perles, Rabin and Shamir in 1963 [20]. Several persons, such as Brzozowski [1] and Friedman [7], have since used the properties of definite machines for feedback studies. The FIFM machine which will be the center of this study is really an extension of the definite machine

to include a feedback.

C. Shift Registers and Feedbacks. Elspas' paper [4] presented an excellent state-of-the-art compilation of linear feedback shift register circuits in 1959. Since then, the work has received much attention and has been expanded by many. Among them are Brzozowski [1] and Friedman [7]. Their results will be the bases for this study. They both discussed single-loop realizations of sequential machines. As was mentioned, properties of definite machines are used in their studies. Friedman showed that all sequential machines can be realized using only one feedback loop and presented techniques of constructing this feedback function.

D. Cellular Circuits. Cellular circuits can actually be identified in designs of contact arrays in the 1930's and diode arrays in 1947. The contemporary investigation of cellular circuits was initiated in 1962 by Maitra in his classical paper "Cascaded Switching Networks of Two-input Flexible Cells" [16]. Unfortunately, the Maitra cascade is not logically complete. Short [22] in 1965 proposed two-rail cascades and these were shown to be complete. Other major contributions are made by Minnick and Short (1964)[17], and Spandorfer and Tonik (1965) [24] in fixed cell arrays and Minnick (1964) [18] in variable cell arrays. A good summary of results in this area with extensive

bibliography can be found in [19]. More recent and advanced theory of cellular logic can be found in a series of SRI reports [5, 6].

II. SM SYNTHESIS USING FIFM PROPERTY

2.1 Introduction

Friedman [7] has shown that all finite-state sequential machines can be realized by circuits with only one feedback. Hence we shall start our discussion by limiting the number of feedbacks to either zero or one. The following questions must be answered for efficient synthesis of a sequential machine.

1. Can we realize the machine without any feedback?
2. If the answer to question 1 is negative, can we use one of the outputs as the feedback?
3. If answers to both questions 1 and 2 are no, how can we construct a function which can be used as the feedback function?

The first two questions will be answered in Section 2.2 by giving testing procedures. The third question will be answered by illustrations in Section 2.3. In Section 2.4 we shall show that the memory K of a machine can be obtained from the tests performed. Two methods of characterizing an FIFM machine are given in Section 2.5. These characterizations will lead us directly to the hardware synthesis of sequential machines. In Section 2.6 we shall study three types of cellular structures. They are the linear cascade, the array, and the tree. Some aspects of minimal realizations and bounds on the number

of cells required will be discussed in Section 2.7. The question of how to initialize a machine to a certain desired state in cellular structures will be ansered in a separate chapter.

2.2 Tests for FIFM

2.2.1 Definiteness Test

Tests to determine whether a machine is feedback free or not are called definiteness tests. We present two definiteness test methods.

A. D-successor Tree Method. A state set is an unordered collection of a finite number of states in which all elements (states) are distinct. Let $A \subseteq S$ be a state set consisting of states $s_1, s_2, \cdots, s_r$, the i_h-successor of the state set A is another state set $B \subseteq S$, formed by next states $f(s_1, i_h)$, $f(s_2, i_h)$, $\cdots$, $f(s_r, i_h)$ with don't cares and repeating states eliminated. A D-successor tree is a tree structure as shown in Figure 4 for M1. The structure is composed of branches arranged in successive levels, the highest level being the zeroth level. In a machine with p input symbols, each branch in the j-th level is plit into p branches in the (j+1)st level. The state sets in the (j+1)st level under the j-th level state set A are i_h-successors of the state set A, where $h = 1, 2, \cdots, p$. Two state sets are said to be connected if they are connected by branches going only upward (or

downward). A state set including all states of a machine is called the universal state set, designated by 1. Hence 1 = S. A state set with no states is a null state set, designated by $\emptyset$.

A D-successor tree always starts with the universal state set. Any state that is not a successor state of any state (including the state itself) of the machine will disappear in the lower levels of the D-successor tree. Each branch of the tree is terminated by one of the following conditions.

1. The state set consists of a single state.
2. The state set is empty (don't cares only).
3. The state set is identical to a connected state set appearing in a lower order level.

Lemma 2.1. A machine M is not definite (i.e., is not feedback free) if and only if at least one of the branches in its D-successor tree is terminated by condition 3.

Proof: Without loss of generality, we assume, as an example, that state set (A, B) is a terminal state set due to condition 3. That is, state set (A, B) repeats itself along a path in the tree. We name this path R. In this case, we can write equations in the form given at the top of page 6 neither for state A nor for state B because an input sequence corresponding to the path R (or multiple repititions of R) could not determine whether the machine is in state A or in state B.

In other words, knowing a finite number of successive inputs is not enough to determine whether the machine is in state A or in state B.

On the other hand, if M is not definite, then at least two states, say C and D, cannot be distinguished by a finite number of successive inputs. This implies that at least one branch in the D-successor tree is terminated by condition 3 with the state set (C, D). For, if it is terminated by condition 1, say with state C, then only state C will have the term corresponding to this path. If it is terminated by condition 2, then the state corresponding to this path is undefined.

Q.E.D.

Theorem 2.2. A machine M is feedback free (definite) if and only if its D-successor tree is terminated by conditions 1 and 2.

Proof: By contra positive of Lemma 2.1

Q.E.D.

Corollary 2.3. If a machine is completely specified and feedback free, its D-successor tree is terminated by condition 1.

Hence, to test whether a machine M is feedback free or not, we construct the D-successor tree of M. A machine is not feedback free as soon as we find a branch in the D-successor tree that is terminated by condition 3. If every branch of the tree is terminated by either conditions 1 or 2, then M is feedback free. Figure 4 shows the D-successor tree of M1. Since some of its branches are terminated by condition 3, M1 is not a feedback free machine.

B. D-pair table Method. A D-pair table is a flow table of the form shown in Figure 5.

1. A column for every input symbol; no output columns are needed.
2. A row for each pair of distinct states s_j and s_k.
3. The entry at the intersection of the i_h-column and the (s_j, s_k)-row is (s_u, s_v) or (s_v, s_u) where $s_u = f(s_j, i_h)$ and $s_v = f(s_k, i_h)$. If $s_u = s_v$ enter only s_u (or s_v).

As an example, the D-pair table of M1 is given in Figure 5. To test whether a given machine is feedback free or not, we follow Algorithm 2.1.

Algorithm 2.1. (D-pair table test)

1. Construct the D-pair table of the machine.
2. Let $k = 0$.
3. Cross out the rows in which all next-state entries are either one of the following types or combinations of them.
   a. don't care,
   b. single state,
   c. a state pair have already been crossed out in an earlier iteration.
4. If not even a single row can be crossed out in step 3, go to step 7; otherwise go to step 5.

5. If all rows of the table have been crossed out, go to step 8; otherwise go to step 6.

6. Let $k = k+1$, go to step 3.

7. The machine is not feedback free. Stop.

8. The machine is feedback free. Stop.

Theorem 2.4. Algorithm 2.1 works.

Proof: The D-pair table method is equivalent to the D-successor tree method going backwards. A crossed-out state pair in the D-pair table represents a pair of states that can be distinguished by a finite input sequence. If every state pair in the D-pair table is crossed out, then every pair of states can be distinguished by a finite number of successive inputs. Hence M is feedback free. Q.E.D.

Figure 5 shows the D-pair table of M1. Since no rows can be crossed out by following the steps of Algorithm 2.1, we conclude that M1 is not feedback free, which agrees with the conclusion of the D-successor tree test.

2.2.2 Finiteness Tests

Tests to determine whether a machine can use one of its output functions as the feedback are called finiteness tests. Since feedback free machines are particular cases of FIFM machines, we would expect that the testing methods presented in the previous section can be

expanded for finiteness tests. This is indeed the case.

A. F-successor Tree Method. Let A be a state set consisting of states $s_1, s_2, \cdots, s_r$, the $\underline{i_h/z_d}$-successors of the state set A are state sets formed by next states $f(s_1, i_h)$, $f(s_2, i_h), \cdots, f(s_r, i_h)$ with equal z_d-output. Thus in a binary machine, there are two i_h/z_d-successors to each state set. Don't cares and repeating states are eliminated in all state sets. An F-successor tree is a tree structure similar to a D-successor tree as shown in Figure 6. Each branch in the j-th level is split into 2p branches in the (j+1)-st level, where p is the number of input symbols. The state sets in the (j+1)-st level under the state set A of the j-th level are i_h/z_d-successors of A. To form an F-successor tree we always start with the universal state set as the zeroth level state set. The conditions for terminating a branch are identical to those given for the D-successor trees. The following properties are analogous to Lemma 2.1 and Theorem 2.2 and are presented without proof.

Lemma 2.5. An output function z_d of a machine M can not be used as the feedback function if and only if at least one of the branches in its F-successor tree is terminated by condition 3.

Theorem 2.6. An output function z_d of a machine M can be used as the feedback function to make M an FIFM machine if and only if its F-successor tree is terminated by conditions 1 and 2.

In summary, if we wish to test whether an output function z_d can be used as the feedback function, we simply construct the F-successor tree for z_d. Conclusions can then be drawn according to the above Lemma and Theorem. Figure 6 shows the F-successor tree of M1. It shows that the output function can be used as the feedback function.

B. F-pair Table Method. An F-pair table for an output z_d is a flow table of the following form.

1. A column for every input symbol; no output columns are needed.
2. A row for each pair of distinct states s_j and s_k.
3. If $g(s_j) = g(s_k)$ for z_d, the entry at the intersection of (s_j, s_k)-row and i_h-column is (s_u, s_v) or (s_v, s_u) where $s_u = f(s_j, i_h)$ and $s_v = f(s_k, i_h)$. If $g(s_j) \neq g(s_k)$ for z_d, the entries at the row (s_j, s_k) are don't cares under all inputs.

As an example, the F-pair table of M1 is given in Figure 7. Algorithm 2.2 describes the procedures of performing the F-pair table test.

Algorithm 2.2.(F-pair Table Test)

1. Let $c = 0$, $d = 1$.
2. Construct the F-pair table for z_d.
3. Let $k = 0$.

4. Cross out all rows in which all next state entries are either one of the following types or combinations of them.

   a. don't care,

   b. single state,

   c. a state pair have already been crossed out in an earlier iteration (smaller k but same d).

5. If not even a single row can be crossed out in step 4, then z_d cannot be used as the feedback function, go to step 10. If all rows have been crossed out, go to step 8; all other cases go to step 7.

7. Let $k = k + 1$, go to step 4.

8. z_d can be used as the feedback function. If we wish to continue tests of other output functions, go to step 9; otherwise stop.

9. Let $c = c + 1$.

10. Is $d = q$? If not, go to step 11; if yes, go to step 12. q is the number of output functions.

11. Let $d = d + 1$, go to step 2.

12. Stop. c = the number of output functions can be used as the feedback function. If $c = 0$, there is no output function can be used as the feedback function.

Theorem 2.7. Algorithm 2.2 works.

Proof: By following the proof of Theorem 2.4. Q.E.D.

Figure 7 shows the F-pair table of M1. Since all rows can be crossed out by carrying out Algorithm 2.2, we conclude that M1 is FIFM by using the output function as the feedback function.

2.3. Converting non-FIFM Machines to FIFM Machines--Construction of the Feedback Function

If we find a machine is not feedback free and also none of its output functions can be used as the feedback function, how can we construct a function such that the machine becomes FIFM? Friedman [7] has answered this question. In this section, we simply present the basic principles involved and illustrate them by simple examples. More detailed treatment can be found in Friedman's paper.

A D-implication graph of a machine M is a state graph constructed according to the D-pair table of M. A loop on a D-implication graph is a closed directed path. If the path does not pass through any node more than once, then the loop is called an elementary loop. A machine M is feedback free if and only if the D-implication graph of M has no elementary loops.

Figure 8a shows the flow table of M3. The D-pair table test in Figure 8b shows that M3 is not feedback free. The F-pair table test in Figure 8c tells us that the output function cannot be used as the feedback function. Hence, we must construct a function, together with the inputs, will make M an FIFM machine. We call this function F.

F is not only an output function but also an input. The D-implication graph of M3 is shown in Figure 8d. There are four elementary loops in that graph. They are

1. $(C, D) \xrightarrow{i_1} (C, D)$

2. $(C, D) \xrightarrow{i_2} (A, C) \xrightarrow{i_1} (C, D)$

3. $(C, D) \xrightarrow{i_2} (A, C) \xrightarrow{i_2} (B, C) \xrightarrow{i_2} (C, D)$

4. $(A, B) \xrightarrow{i_2} (B, D) \xrightarrow{i_2} (A, D) \xrightarrow{i_2} (A, B)$

Since it is the elementary loops that are causing the trouble, we wish to add an additional input, F, such that all elementary loops become non-loop. To do so, at least one state pair in each loop should have different values of F for the two states. For example, if we assign 0 as the F value for state C and 1 for state D, then the state pair (C, D) will go to state C under input i_1-0 (original input plus F) and will go to state D under input i_1-1. Thus eliminate the elementary loop (1). Hence, to break the elementary loop (1), state C and D should have different F values. Similarly, to break the elementary loop (2), either states C and D or states A and C should be assigned different F values. Similar reasonings apply to the other two loops. If we use the notations

A_1 = state A has assigned F value 1,

A_0 = state A has assigned F value 0,

$$A_1 B_0 = A_1 \text{ and } B_0,$$

$$A_1 + B_0 = A_1 \text{ or } B_0,$$

then, we can write the requirements for F in a more compact form.

To break loop (1), we need $C_0 D_1 + C_1 D_0$;

to break loop (2), we need $C_0 D_1 + C_1 D_0 + A_0 C_1 + A_1 C_0$;

to break loop (3), we need $C_0 D_1 + C_1 D_0 + A_0 C_1 + A_1 C_0 + B_0 C_1 + B_1 C_0$;

to break loop (4), we need $A_0 B_1 + A_1 B_0 + B_0 D_1 + B_1 D_0 + A_0 D_1 + A_1 D_0$.

F must satisfy all above criteria. Hence,

$$\begin{aligned} F = {} & (C_0 D_1 + C_1 D_0)(C_0 D_1 + C_1 D_0 + A_0 C_1 + A_1 C_0) \\ & (C_0 D_1 + C_1 D_0 + A_0 C_1 + A_1 C_0 + B_0 C_1 + B_1 C_0) \\ & (A_0 B_1 + A_1 B_0 + B_0 D_1 + B_1 D_0 + A_0 D_1 + A_1 D_0). \end{aligned} \tag{2.1}$$

By using the obvious properties that

$$A_0 A_1 = 0, \qquad A_0 + A_1 = 1,$$

$$A_1 A_1 = A_1, \qquad A_1 + A_1 = A_1,$$

$$A_0 A_0 = A_0, \qquad A_0 + A_0 = A_0,$$

$$0X = 0, \qquad 1 + X = 1,$$

where $X = A_0$ or A_1

We can simplify Equation 2.1 to

$$\begin{aligned} F = {} & (C_0 D_1 + C_1 D_0)(1 + A_0 C_1 + A_1 C_0(1 + B_0 C_1 + B_1 C_0)) \\ & (A_0 B_1 + A_1 B_0 + B_0 D_1 + B_1 D_0 + A_0 D_1 + A_1 D_0) \end{aligned}$$

$$= (C_0D_1 + C_1D_0)(A_0B_1 + A_1B_0 + B_0D_1 + B_1D_0 + A_0D_1 + A_1D_0)$$

$$= B_1C_1D_0 + B_0C_0D_1 + A_1C_1D_0 + A_0C_0D_1 \ . \qquad (2.2)$$

Hence, F can be obtained by choosing any one of the following assignments:

a. $B_1C_1D_0$, i.e., 1--B, 1--C, 0--D, A can be either 0 or 1,

b. $B_0C_0D_1$, i.e., 0--B, 0--C, 1--D, A can be either 0 or 1,

c. $A_1C_1D_0$, i.e., 1--A, 1--C, 0--D, B can be either 0 or 1,

d. $A_0C_0D_1$, i.e., 0--A, 0--C, 1--D, B can be either 0 or 1.

Note that whatever assignments we choose, the F values are associated with states and can be obtained by making F an output function of the machine. Figure 9a shows M3' (M3 with an added output F). The F-pair table test of M3' using F as the feedback function is shown in Figure 9b. It shows that M3' is an FIFM machine. Thus, F is a valid feedback function.

There are cases when a simple feedback function cannot be found because of conflicting conditions required to resolve all elementary loops. In these cases, some or all states have to be split. The process of state splitting is given in [7].

Sometimes when the D-implication graph is complicated, it is rather difficult to find all elementary loops. D-pair trees may be helpful in this respect. A D-pair tree is like a D-successor tree except that it is constructed according to the D-pair table and the

zeroth level state set is a state pair. To find all elementary loops, we construct a D-pair tree for every state pair of the machine. Any branch which is terminated by condition 3 represents an elementary loop (not necessarily different from the other elementary loops). As an example, the D-pair trees of M3 are given in Figure 10. Though all four types of elementary loops are evident in the first D-pair tree, in general we must construct all D-pair trees of M to make sure that there are no other loops. Of course, experience can always cut a great amount of unnecessary work.

2.4. Determination of K

In this section, we shall relate the memory K of a machine M to the testing methods of earlier sections.

Theorem 2.8. The memory K of a feedback free machine M is equal to the last level number, L, in a longest branch of the D-successor tree. i.e., K = L.

Proof: Without loss of generality, we assume that the state set along a longest branch in the (L-1)st level is (A, B). Then, one of the following conditions must be true for $h = 1, 2, \cdots, p$.

a. $f(A, i_h) = f(B, i_h)$ or

b. $f(A, i_h)$ and/or $f(B, i_h)$ are undefined.

If not, then the tree would either have (L+1)-st level or is terminated by condition 3 --- both contradict our assumptions. Therefore, L

past inputs together with the present input are sufficient to determine the present state of M. It is obvious that L is the minimal number of successive past inputs required. Hence, K = L. Q.E.D.

Theorem 2.9. The memory K of a feedback free machine M is equal to the final k value in carrying out the D-pair table test. Let N be the final k value, then K = N.

Proof: By similar arguments given above. Q.E.D.

Theorem 2.10. The memory K of an FIFM machine with z_d as the feedback function is equal to the total number of levels in a longest branch of the F-successor tree for z_d. i.e., K = L + 1.

Proof: Similar to that of Theorem 2.8. Since the feedback obtained at t-1 is used as an input at t, the feedback input at t-L is the feedback obtained at t-(L+1). In other words, the feedback input has one more delay than the original inputs. Hence, K = L + 1. Q.E.D.

Theorem 2.11. The memory K of an FIFM machine M with z_d as the feedback is equal to the total number of iterations required in carrying out the F-pair table test for z_d. i.e., K = N + 1.

Proof: Obvious. Q.E.D.

If an added function is used as the feedback function, we simply consider it as an output function z_d and K can then be found. From the above results, we can see that if different output functions

are used as the feedback function, we may have different memory K. For economy (see Section 2.7), we usually choose the output function that will give the minimal K as the feedback function. By the same reason, we always select the feedback function, among all possibilities, the one will give minimal K. This is done, more or less, by trial and compare method.

2.5. Characterization of FIFM Machines

In this section we shall introduce two methods of characterizing a machine using the results developed earlier in this chapter. They will lead us to easy construction of sequential circuits in the subsequent section.

2.5.1 Difference Equations

When time also becomes a factor in a Boolean equation, it is called a difference equation. We define

$$\Delta^0 f(t) = f(t).$$

$$\Delta f(t) = \Delta^1 f(t) = f(t-1).$$

$$\Delta^m f(t) = f(t-m).$$

$$\Delta^{-m} f(t) = f(t+m).$$

Δ is called the difference operator. Examples:

$$\Delta^{0}(x_t + y_{t-1} + x_{t-2}z_t) = x_t + y_{t-1} + x_{t-2}z_t .$$

$$\Delta(x_t + y_t + x_{t-1}y_{t-1}) = x_{t-1} + y_{t-1} + x_{t-2}y_{t-2}$$

$$\Delta^{3}(x_t) = x_{t-3}$$

$$\Delta^{-2}(x_{t-2} + y_{t-3} + z_{t-4}) = x_t + y_{t-1} + z_{t-2}$$

Some direct and obvious properties of the difference operator are given below.

$$\Delta C = C, \quad C \text{ is a constant.}$$

$$\Delta[\Delta f(t)] = \Delta^{2} f(t).$$

$$\Delta^{m}[\Delta^{n} f(t)] = \Delta^{n}[\Delta^{m} f(t)] = \Delta^{m+n} f(t).$$

$$\Delta^{m}[f(t) + g(t)] = \Delta^{m} f(t) + \Delta^{m} g(t).$$

$$\Delta^{m}[\{f(t)\}\{g(t)\}] = \{\Delta^{m} f(t)\}\{\Delta^{m} g(t)\}.$$

$$\Delta^{m}[f(t) + g(t)] = \Delta^{m} f(t) + \Delta^{m} g(t).$$

$$\Delta^{m}\{[f(t)]'\} = \{\Delta^{m} f(t)\}'$$

$$\{[\Delta^{m} f(t)][\Delta^{n} g(t)]\}' = [\Delta^{m} f(t)]' + [\Delta^{n} g(t)]'$$

$$\{\Delta^{m} f(t) + \Delta^{n} g(t)\}' = [\Delta^{m} f(t)]'[\Delta^{n} g(t)]'$$

We now show that how an FIFM machine can be described by difference equations. M2 with X as the input variable is shown in Figure 11a. The D-pair table test in Figure 11b shows that M2 is a

feedback free machine with $K = 3$. Hence, we would expect that the machine can be described by the present input and three successive past inputs.

Note that the output of M2 is 1 when it is in states B, C or E. We associate our output Z with the state set (BCE).

$$Z_t = (BCE)_t \tag{2.3}$$

Also note that M2 is in the state set (BCE) only if it was (assuming zero transition time)

1. in the state set (BCD) and the input is $\overline{X}$, or
2. in the state set (CE) and the input is X.

In our notation,

$$(BCE)_t = \overline{X}_t \Delta(BCD)_t + X_t \Delta(CE)_t .$$

By similar reasonings,

$$(BCD)_t = \overline{X}_t \Delta(BD)_t + X_t \Delta(ABCDE)_t = \overline{X}_t \Delta(BD)_t + X_t \Delta 1$$

$$= \overline{X}_t \Delta(BD)_t + X_t,$$

$$(CE)_t = \overline{X}_t \Delta(BCD)_t,$$

$$(BD)_t = \overline{X}_t \Delta(ABCDE)_t = X_t \Delta 1 = X_t.$$

Note that we use 1 for the universal state set and 0 for the empty set. Hence,

$$Z_t = (BCE)_t$$

$$= \overline{X}_t \Delta(BCD)_t + X_t \Delta(CE)_t$$

$$= \overline{X}_t \Delta \{ \overline{X}_t \Delta (BD)_t + X_t \} + X_t \{ \overline{X}_t \Delta (BCD)_t \}$$

$$= \overline{X}_t \Delta \{ \overline{X}_t \Delta (X_t) + X_t \} + X_t \Delta \{ \overline{X}_t \Delta [\overline{X}_t \Delta (BD)_t + X_t] \}$$

$$= \overline{X}_t \Delta \{ \overline{X}_t \Delta (X_t) + X_t \} + X_t \Delta \{ \overline{X}_t \Delta [\overline{X}_t \Delta (X_t) + X_t] \}$$

$$= \overline{X}_t \Delta \overline{X}_t \Delta^2 (X_t) + \overline{X}_t \Delta X_t + X_t \Delta \overline{X}_t \Delta^2 \overline{X}_t \Delta^3 X_t + X_t \Delta \overline{X}_t \Delta^2 X_t$$

$$= \overline{X}_t \overline{X}_{t-1} X_{t-2} + \overline{X}_t X_{t-1} + X_t \overline{X}_{t-1} \overline{X}_{t-2} X_{t-3} + X_t \overline{X}_{t-1} X_{t-2} \qquad (2.4)$$

As another example, M1 is again given in Figure 12a. It was shown that M1 is FIFM using the output as the feedback. Since the feedback is used as another input, we expand the original flow table to that shown in Figure 12b. The next state entries for impossible combinations of inputs are don't cares, e.g., if the machine is in state A, then the Z value has to be 1. Hence the next state entries of A in the columns with $\overline{F}$ (where F_t is the same as Z_{t-1}) are don't cares.

$$Z_t = (AC)_t$$

$$(AC)_t = \overline{X}_t \overline{Z}_{t-1} \Delta (D \odot AC)_t + \overline{X}_t Z_{t-1} \Delta (A \odot BD)_t + X_t \overline{Z}_{t-1} \Delta (D \odot AC)_t + X_t Z_{t-1} (C \odot BD)_t$$

Notation: $\odot$ → The term in front of $\odot$ is required while the term after $\odot$ is don't care and can either be included or not included as desired.

$$
\begin{aligned}
(D \odot AC)_t &= \bar{X}_t \bar{Z}_{t-1} \Delta(ABCD)_t + \bar{X}_t Z_{t-1} \Delta(ABCD)_t + X_t \bar{Z}_{t-1} \Delta(ABCD)_t \\
&\quad + X_t Z_{t-1} \Delta(0) \\
&= \bar{X}_t \bar{Z}_{t-1} + \bar{X}_t Z_{t-1} + X_t \bar{Z}_{t-1} \\
(A \odot BD)_t &= \bar{X}_t \bar{Z}_{t-1} \Delta(0) + \bar{X}_t Z_{t-1} \Delta(0) + X_t \bar{Z}_{t-1} \Delta(ABCD)_t + X_t Z_{t-1} \Delta(0) \\
&= X_t \bar{Z}_{t-1} \\
(C \odot BD) &= \bar{X}_t \bar{Z}_{t-1} \Delta(ABCD)_t + \bar{X}_t Z_{t-1} \Delta(ABCD)_t + X_t \bar{Z}_{t-1} \Delta(0) \\
&\quad + X_t Z_{t-1} \Delta(ABCD)_t \\
&= \bar{X}_t \bar{Z}_{t-1} + \bar{X}_t Z_{t-1} + X_t Z_{t-1} \\
Z_t &= (AC)_t \\
&= \bar{X}_t \bar{Z}_{t-1} \Delta(D \odot AC)_t + \bar{X}_t Z_{t-1} \Delta(A \odot BD)_t \\
&\quad + X_t \bar{Z}_{t-1} \Delta(D \odot AC)_t + X_t Z_{t-1} (C \odot BD)_t \\
&= \bar{X}_t \bar{Z}_{t-1} \Delta(\bar{X}_t \bar{Z}_{t-1} + \bar{X}_t Z_{t-1} + X_t \bar{Z}_{t-1}) + \bar{X}_t Z_{t-1} \Delta(X_t \bar{Z}_{t-1}) \\
&\quad + X_t \bar{Z}_{t-1} \Delta(\bar{X}_t \bar{Z}_{t-1} + \bar{X}_t Z_{t-1} + X_t \bar{Z}_{t-1}) \\
&\quad + X_t Z_{t-1} \Delta(\bar{X}_t \bar{Z}_{t-1} + \bar{X}_t Z_{t-1} + X_t Z_{t-1}) \\
&= \bar{X}_t \bar{Z}_{t-1} \bar{X}_{t-1} \bar{Z}_{t-2} + \bar{X}_t \bar{Z}_{t-1} \bar{X}_{t-1} Z_{t-2} + \bar{X}_t \bar{Z}_{t-1} X_{t-1} \bar{Z}_{t-2} \\
&\quad + \bar{X}_t Z_{t-1} X_{t-1} \bar{Z}_{t-2} + X_t \bar{Z}_{t-1} \bar{X}_{t-1} \bar{Z}_{t-1} + X_t \bar{Z}_{t-1} \bar{X}_{t-1} Z_{t-2} \\
&\quad + X_t \bar{Z}_{t-1} X_{t-1} \bar{Z}_{t-2} + X_t Z_{t-1} \bar{X}_{t-1} \bar{Z}_{t-2} + X_t Z_{t-1} \bar{X}_{t-1} Z_{t-2} \\
&\quad + X_t Z_{t-1} X_{t-1} Z_{t-2}
\end{aligned}
\tag{2.5}
$$

As the third example, we write the characterizing equations for M3 where F is the constructed feedback function. The expanded flow table with the feedback as one of the inputs is shown in Figure 13b.

$$Z_t = (CD)_t \tag{2.6}$$

$$F_t = (AC)_t \tag{2.7}$$

$$(CD)_t = \bar{X}_t\bar{F}_{t-1}\Delta(BD \odot AC)_t + \bar{X}_tF_{t-1}\Delta(AC \odot BD)_t + X_t\bar{F}_t\Delta(B \odot AC)_t + X_tF_{t-1}\Delta(C \odot BD)_t$$

$$(B \odot AC)_t = \bar{X}_t\bar{F}_{t-1}\Delta(0) + \bar{X}_tF_{t-1}\Delta(0) + X_t\bar{F}_{t-1}\Delta(0) + X_tF_{t-1}\Delta(ABCD)_t$$

$$= X_tF_{t-1}$$

$$(C \odot BD)_t = \bar{X}_t\bar{F}_{t-1}\Delta(ABCD)_t + \bar{X}_tF_{t-1}(ABCD)_t + X_t\bar{F}_{t-1}\Delta(0) + X_tF_{t-1}\Delta(ABCD)_t$$

$$= \bar{X}_t\bar{F}_{t-1} + \bar{X}_tF_{t-1} + X_tF_{t-1}$$

$$Z_t = (CD)_t$$

$$= \bar{X}_t\bar{F}_{t-1}\Delta(BD \odot AC)_t + \bar{X}_tF_{t-1}\Delta(AC \odot BD)_t + X_t\bar{F}_{t-1}\Delta(B \odot AC)_t + X_tF_{t-1}\Delta(C \odot BD)_t$$

$$= \bar{X}_t\bar{F}_{t-1} + \bar{X}_tF_{t-1} + X_t\bar{F}_{t-1}\Delta(X_tF_{t-1}) + X_tF_{t-1}\Delta(\bar{X}_t\bar{F}_{t-1} + \bar{X}_tF_{t-1} + X_tF_{t-1})$$

$$= \bar{X}_t\bar{F}_{t-1} + \bar{X}_tF_{t-1} + X_t\bar{F}_{t-1}X_{t-1}F_{t-2} + X_tF_{t-1}\bar{X}_{t-1}\bar{F}_{t-2}$$
$$+ X_tF_{t-1}\bar{X}_{t-1}F_{t-2} + X_tF_{t-1}X_{t-1}F_{t-2} \tag{2.8}$$

Similarly,

$$F_t = \bar{X}_t\bar{F}_{t-1}\bar{X}_{t-1}\bar{F}_{t-2} + \bar{X}_t\bar{F}_{t-1}\bar{X}_{t-1}F_{t-2} + \bar{X}_t\bar{F}_{t-1}X_{t-1}\bar{F}_{t-2}$$
$$+ \bar{X}_tF_{t-1}X_{t-1}\bar{F}_{t-2} + X_t\bar{F}_{t-1}\bar{X}_{t-1}\bar{F}_{t-2} + X_t\bar{F}_{t-1}\bar{X}_{t-1}F_{t-2}$$
$$+ X_t\bar{F}_{t-1}X_{t-1}\bar{F}_{t-2} + X_tF_{t-1}\bar{X}_{t-1}\bar{F}_{t-2} + X_tF_{t-1}\bar{X}_{t-1}F_{t-2}$$
$$+ X_tF_{t-1}X_{t-1}F_{t-2} \tag{2.9}$$

The techniques described in this section are completely general and can be used for machines with multiple inputs and outputs. The difference equations thus obtained, however, may be long and it is easy to make mistakes in the calculations. We now introduce a more compact way of characterizing a sequential machine.

2.5.2 Predecessor Trees

When $f(s_j, i_h) = s_k$; s_j is said to be the immediate predecessor state of s_k under i_h. We use the notation $s_j = f^{-1}(s_k, i_h)$. Let A be a state set consisted of states $s_1, s_2, \cdots, s_r$, the i_h-predecessor of A is a state set formed by immediate predecessor states $f^{-1}(s_1, i_h)$, $f^{-1}(s_2, i_h)$, $\cdots$, $f^{-1}(s_r, i_h)$. The immediate predecessor states of don't care states can be included, if desired, in the i_h-predecessor

of A. The z_d-predecessor tree is a tree structure as shown in Figure 14. The structure is composed of branches arranged in successive levels, the highest level being the zeroth level and is a state set composed of states with $g(s) = 1$ for z_d. In a machine with p input symbols (including the feedback input), each branch in the j-th level is split into p branches in the (j+1)-st level. The state set in the (j+1)-st level under a j-th level state set A and a branch i_h is the i_h-predecessor of A. A branch is terminated by meeting one of the following conditions.

1. A state set consisted of all states, (1)
2. A state set consisted of no states, (∅)
3. A state set repeats a connected state set appeared in an earlier level.

A machine that has identical outputs for all states or has nonzero outputs for transient states only is a trivial machine.

Lemma 2.12. For a nontrivial feedback free machine M, at least one branch of its predecessor tree is terminated by condition 1.

Proof: A nontrivial machine has at least one nontransient state, say state B, in the zeroth level state set of its predecessor tree. By definition, a feedback free machine is a machine which can start on any state and be led to any nontransient state by an input sequence. This is the case for state B. Hence, there must be a branch in the

tree, corresponding to that input sequence, that is terminated by condition 1. Q. E. D.

Theorem 2.13. The predecessor tree of a feedback free machine M is terminated by conditions 1 and/or 2.

Proof: Necessity: 1. There are branches which are terminated by condition 1 as was discussed in Lemma 2.12. 2. There are states under a specified input does not have any predecessor states. If every state in a state set has no predecessor under i_h, then the i_h branch under that state set is empty and is terminated by condition 2.

Sufficiency: We simply show that whenever a branch of a predecessor tree is terminated by condition 3, then M is not a feedback free machine. Without loss of generality, we assume that a state set (A, B) repeats itself along a certain path of the tree. If we consider this path in reverse order, it is exactly an elementary loop in a D-pair tree (or D-implication graph). Hence, M is not feedback free. Q. E. D.

If in an FIFM machine, we consider F as an output and the feedback input as another input, the expanded machine, e.g., Figure 13b, can be thought as feedback free. Hence, an FIFM machine can also be characterized by predecessor trees. Examples used in Section 2.5.1 are again used in Figures 14--16 to illustrate this characterization technique.

Note that the reasoning behind the construction of predecessor trees is exactly the same as that of difference equations. Hence, there are close relations between the two. If we consider the branch between the zeroth level and the first level corresponding to the input at time t and the feedback at t-1, those between the first level and the second level corresponding to the input at time t-1 and the feedback at t-2, etc., we can easily write the difference equations for a given machine from its predecessor trees. The difference equation consists of a sum of product terms. Each product term is formed by the product of the terminal condition and the path leading to that terminal from the zeroth level. For example, the difference equation of M1 can be obtained from Figure 15b.

$$
\begin{aligned}
Z_t = {} & \bar{X}_t \bar{Z}_{t-1} (1 \cdot \bar{X}_{t-1} \bar{Z}_{t-2} + 1 \cdot \bar{X}_{t-1} Z_{t-2} + 1 \cdot X_{t-1} \bar{Z}_{t-2} + \phi \cdot X_{t-1} Z_{t-2}) \\
& + \bar{X}_t Z_{t-1} (\phi \cdot \bar{X}_{t-1} \bar{Z}_{t-2} + \phi \cdot \bar{X}_{t-1} Z_{t-2} + 1 \cdot X_{t-1} \bar{Z}_{t-2} + \phi \cdot X_{t-1} Z_{t-2}) \\
& + X_t \bar{Z}_{t-1} (1 \cdot \bar{X}_{t-1} \bar{Z}_{t-2} + 1 \cdot \bar{X}_{t-1} Z_{t-2} + 1 \cdot X_{t-1} \bar{Z}_{t-2} + \phi \cdot X_{t-1} Z_{t-2}) \\
& + X_t Z_{t-1} (1 \cdot \bar{X}_{t-1} \bar{Z}_{t-2} + 1 \cdot \bar{X}_{t-1} Z_{t-2} + \phi \cdot X_{t-1} \bar{Z}_{t-2} + 1 \cdot X_{t-1} Z_{t-2}) \\
= {} & \bar{X}_t \bar{Z}_{t-1} \bar{X}_{t-1} \bar{Z}_{t-2} + \bar{X}_t \bar{Z}_{t-1} \bar{X}_{t-1} Z_{t-2} + \bar{X}_t \bar{Z}_{t-1} X_{t-1} \bar{Z}_{t-2} \\
& + \bar{X}_t Z_{t-1} X_{t-1} \bar{Z}_{t-2} + X_t \bar{Z}_{t-1} \bar{X}_{t-1} \bar{Z}_{t-2} + X_t \bar{Z}_{t-1} \bar{X}_{t-1} Z_{t-2} \\
& + X_t \bar{Z}_{t-1} X_{t-1} \bar{Z}_{t-2} + X_t Z_{t-1} \bar{X}_{t-1} \bar{Z}_{t-2} + X_t Z_{t-1} \bar{X}_{t-1} Z_{t-2} \\
& + X_t Z_{t-1} X_{t-1} Z_{t-2} \cdot
\end{aligned}
$$

This is exactly the same as Equation 2.5 derived in the previous section. Note that unnecessary work can be eliminated by simply writing down the branches terminated by (1) and summing them up.

2.6 Cellular Synthesis

Since all sequential machines can be described by combinational type equations, they can be realized by conventional combinational circuits if variables of different time subscripts are all available at the same time. This can be easily achieved by delay elements. Because the number of input variables and the memory K are both finite, the required number of delay elements is also finite.

Prior to the advent of integrated circuit technology, the goal of every logic designer was concentrated around minimizing the number of individual components in hardware realizations. But with the new technology, aside from a given upper limit, a large percentage of the cost is independent of the complexity of each building block. The main objective is, instead, to minimize the number of building blocks (modules). The more uses that can be found for a given module, the less expensive the per unit cost will be. Hence, to adapt large scale integration into future machines,

1. the number of types of modules must be small, and
2. the interconnection of modules to form larger structures must be simple and regular.

To meet the first criterion, we limit our cell types to just one. That is, all cells are identical. To meet the second criterion, we investigate the feasibility of three different cellular structures. They are the linear cascade, the array, and the tree.

2.6.1 Linear Cascade

There are many ways to make linear cascades. We shall limit our investigation to that of unilatral linear cascades. The earliest cascade is the Maitra cascade [16] as shown in Figure 17. (These are called tributary switching network by Sklansky). The Maitra cascade, however, is not logically complete even if one allows redundant cascade (Stone and Korenjak [25]) in which the same external input may be connected to inputs of several cells. As pointed out by Short [22], in order to make the linear cascade functionally complete, the only logical direction is to expand the number of the interconnecting leads between cells. He showed that the two rail cascade is functionally complete. As an example, the realization of M2 using the two rail cascade is shown in Figure 18.

There are two ways to obtain necessary delays for the inputs. All delay elements are assumed to be unit delay elements in our study.

1. Delayed input: inputs are delayed at different stages as shown in Figure 19a.

2. Delayed logic: delays are distributed throughout the logic circuits as shown in Figure 19b.

The circuit of Figure 19b has one more delay than that of Figure 19a and the output is actually

$$Z_t = \Delta(BCE)_t = \Delta(\bar{X}_t\bar{X}_{t-1}X_{t-2} + \bar{X}_tX_{t-1} + X_t\bar{X}_{t-1}\bar{X}_{t-2}X_{t-3} + X_t\bar{X}_{t-1}X_{t-2})$$

$$= \bar{X}_{t-1}\bar{X}_{t-2}X_{t-3} + \bar{X}_{t-1}X_{t-2} + X_{t-1}\bar{X}_{t-2}\bar{X}_{t-3}X_{t-4} + X_{t-1}\bar{X}_{t-2}X_{t-3}.$$

Thus, when delayed input circuit is used, we are assuming the output is obtained with zero time. When delayed logic circuit is used, the output is obtained in one clock time. The advantage of delayed input is that it uses less delay elements while that of delayed logic is more uniform in structure.

The two rail cascade, however, can realize only one function per cascade. In many sequential machine syntheses, we need to realize both the output function and the feedback function not to mention multiple output functions. Although some studies in multiple function realization using two rail cascades have been done [3, 28], it is complicated and not efficient for general sequential machine realization. Of course, we can always realize multiple functions by multiple two rail cascades. This sort of approach comes very close to the array

type circuit we will discuss in the next section.

If we expand the number of the interconnecting leads further, we have the general multiple-rail cascade. But if the cell is to remain relatively simple, the number of rails that we may have must remain small. The difficulties of realizing multiple functions encountered in the two-rail cascade will also be present in this case.

Now we consider the synthesis from the other direction--that of machine decomposition. Krohn and Rhodes [14] stated that each finite state machine M can be built as a cascade connection of two-state machine and permutation machines. Zeiger [29], in studying the cascaded P-R machine synthesis of sequential machines, also proved that this is possible. This is done by further decomposition of P-R machines into two-state machines and permutation machines. (Note that a two-state machine is a P-R machine.) However, in this kind of cascade, the number of different cell types can be very large (there are infinite number of P-R machines) and does not meet our criterion of single cell type.

Therefore, we conjecture that, at the present, there is no linear cascade which is general enough to realize all finite state sequential machines with just a single cell type and yet the cell structure is quite simple.

2.6.2 Array

The natural direction for extending the linear cascade is to the two dimensional array. Several cellular array structures for combinational logic have been studied and are discussed in Appendix II. Since we can write all characterizing equations as the combinational results of a finite number of successive inputs and feedbacks, all combinational cellular arrays can be used to synthesize sequential machines by slight modifications--addition of delay elements. For example, Figure 20 shows the cellular realization of the function

$$W = \bar{X}_0\bar{X}_1X_2 + \bar{X}_0X_1 + X_0\bar{X}_1\bar{X}_2X_3 + X_0\bar{X}_1X_2 \qquad (2.12)$$

using Spandorfer's technique [24].

Recall that the output equation for M2 was found to be

$$Z_t = \bar{X}_t\bar{X}_{t-1}X_{t-2} + \bar{X}_tX_{t-1} + X_t\bar{X}_{t-1}\bar{X}_{t-2}X_{t-3} + X_t\bar{X}_{t-1}X_{t-2}. \qquad (2.13)$$

If we make the following correspondances,

$$Z_t = W,\ X_t = X_0,\ X_{t-1} = X_1,\ X_{t-2} = X_2,\ X_{t-3} = X_3 \text{ and } X_{t-4} = X_4,$$

then Equation 2.12 and Equation 2.13 are identical. Figure 21a shows the delayed input realization of M2. Figure 21b is the delayed logic realization of M2.

As another example, the realization of M3 is shown in Figure 22.

The above are examples of fixed cell-function arrays, i.e., the functional performance of each cell is fixed. When cell functions can be varied over some useful set of possible functions, the structure is called variable cell-function array. One such array is known as the cutpoint cellular array. In a cutpoint cellular array, each cell is able to perform one of the nine specified functions. These functions are listed in Figure 23 where x and y are inputs and z is the output. The number N indicates the function the cell is to perform. Originally, a specific function is achieved by cutting certain circuit connections within the cell, hence the name cutpoint. Figure 24 shows the cutpoint cellular array synthesis of M2.

The synthesis of M1 (a machine with the feedback) using cutpoint cellular array with delayed input is shown in Figure 25.

From above examples, we can see that once the difference equations were obtained, the sequential machine can be easily realized by cellular arrays.

2.6.3 Tree

In combinational logic, the tree represents a special case of design structure and has received a good deal of attention. A general tree of n variables has 2^n terminal nodes and each represents an n-variable minterm. Any n-variable function can be realized by combining appropriate minterms. For example, Figure 26a realizes the

function

$$W = \overline{X}_0\overline{X}_1X_2 + X_0\overline{X}_1 + X_0X_1X_2 . \qquad (2.14)$$

The tree structure of Figure 26a can be composed of either relays or gates. Consider the case of using relays (which are bidirectional), we can turn around and get the output from the tip of the tree as shown in Figure 26b. If we call the left terminals the boundary terminals, then Figure 26b shows that all functions can be realized by giving proper boundary conditions.

Figure 27 shows the same two ways of realization of Equation 2.14 using gates. From Figure 27b, we can see that all functions can be realized by giving proper boundary conditions. Note the uniform structure in Figure 27b. To simplify the work, we use the symbol shown in Figure 28b to represent the gate network of Figure 28a and use the symbol in Figure 29b to represent the gate network of Figure 29a.

Thus, a sequential machine can easily be realized by tree circuits once the difference equations for the machine are obtained. Figure 30 shows the tree realizations of M2. For uniformity, we demand that all branches in the tree have an equal number of levels, even though some of them could otherwise be shortened.

Before we discuss how the feedback input, the multiple inputs and the multiple output are implemented, we now relate the realization

to the predecessor tree. Recall that a predecessor tree also describes an output function. If we extend all branches of the predecessor tree to K-th level, we have a uniform tree. Note that all branches of a predecessor tree must be terminated either by (1) or by ($\emptyset$). For branches terminated before the K-th level with a (1), we extend it to the K-th level with all (1)'s. For branches terminated before the K-th level with a ($\emptyset$), we extend it to the K-th level with all ($\emptyset$)'s. The K-th level in the predecessor tree provides us the proper boundary conditions for the tree realization. Of course, we must be careful to make the correct correspondences between the branches of the predecessor tree and those of the cellular tree. This enables us to synthesize any sequential machine in tree form directly from its predecessor trees. When multiple output functions are required, we simply use one tree for each output function. If the machine has multiple inputs (including the feedback input), we use a multiple "layer" for each level as illustrated in Figure 31c. Thus, for notational convenience, if there are p-inputs, we shall combine 2^p-1 cells and call it a <u>super cell.</u> Examples of super cell symbols are given in Figure 32. Note that if we use the notation of super cells, the realization tree will have the similar structural form as the extended predecessor tree in all cases.

2.7 Minimal Realizations and Bounds

A. Tree Structure. In the tree structure, many cells are inactive and can be eliminated. But for uniformity, we demand that all branches have equal length. Hence, the number of cells required is

$$N = q(2^{p(K+1)}-1), \tag{2.15}$$

where p = total number of input variables (including the feedback input if any),

q = total number of output functions (including the feedback function if any), and

K = memory of the machine.

If each super cell is considered as one unit, then

$$N = q(2^{K+1}-1) \tag{2.16}$$

If the machine is feedback free, we must study the tradeoff problem between K and p and q. If the decrease in K more then offsets the increase in p and q by using a feedback, then it is more economical to have the feedback. Since we restrict our study to that of only a single feedback, the trade off between multiple feedback and p and q will not be discussed. Of course, we always choose the feedback which gives the minimal K, hence the minimal N. In Figure 33, the numbers on slanted lines indicate the numbers of K that can be decreased by having the feedback. The region above that slanted line

indicates the combination of p and K that will give lower N by having the feedback. For FIFM machines, it is always more economical to use the feedback function that will give minimal K.

B. Array Structure. The number of cells required will depend on the type of array being used. But for both types of arrays discussed in the previous section, the process involves realizing the product terms and then summing them. If we consider four NAND gates as a single cell in our fixed function arrays, then the number of cells required for both fixed and variable function arrays are

$$N = (pK + 1 + w)(m_1 + m_2 + \cdots + m_q). \qquad (2.17)$$

where p = total number of input variables (including the feedback input, if any),

q = total number of output functions (including the feedback function, if any),

K = the memory of the machine,

m_i = number of product terms in the i-th output function,

w = 0, if there is a feedback, and

= 1, if there is no feedback.

Since

$$m_i \leq 2^{p\,K+1} + w,$$

$$\sum_{i=1}^{q} m_i \leq q \cdot 2^{p\,K+1} + w.$$

Hence,

$$N \leq [p\ K+1\ + w](q \cdot 2^{p(K+1)} + w). \qquad (2.18)$$

The trade-off problem, however, is not as simple. For most cases, a minimal K would mean a minimal realization. But this is not necessarily true for all cases since a minimal K may have large number of minterms for output functions that would make N large. The conjecture is that there is no general optimal solution for this case. For the lack of such solution, we would assume that the realization is minimal if K is minimal. Of course, for any chosen K, we would always try to minimize the number of terms in each output function using normal combinational techniques.

2.8 Summary

For any given machine M, we first test whether M can be realized without any feedback. Two methods, the D-successor tree and the D-pair table, are available by feedback free circuits. The next step is to test whether one of its output functions can be used as the feedback function. This can be done by one of the two finiteness test methods--F-successor tree and F-pair table. In both the definiteness test and finiteness test, the pair table method is preferred because of its simplicity and its necessity in constructing the D-implication graph if needed. The D-implication graph is used as an aid to determine the feedback function required if M fails both the definiteness

test and the finiteness test. The successor tree method is introduced because it is a straight forward interpretation of the definition of FIFM machines. It also introduces the notion of a tree structure. D-pair tree essentially break the D-implication graph into several sub-graphs. They help to identify elementary loops when the D-implication graph is too complicated. The memory K of a machine can also be found as a side result of the tests.

Two mathematical tools that can be used to characterize a given sequential machine were developed in Section 2.5. The difference equation obtained by using difference operator is mathematically more vigorous while the predecessor tree is much simpler to construct. If a machine M is feedback free, we can use one of the two methods to describe its output functions in terms of a finite number of successive inputs. If a machine M is FIFM, we first obtain the expanded flow table of M and then use one of the two methods to describe its output and feedback functions in terms of a finite number of successive inputs and feedbacks. In other words, all sequential machines can be characterized by combinational equations if we consider a variable with different time subscripts as different variables. We then studied the characteristics of several cellular structures that are appropriate for synthesizing finite-state sequential machines. The restriction we imposed are that 1. cells must be identical; 2. cells must be relatively simple; 3. intercell connections must be uniform except for the

final synthesis connections. After investigating several possible linear cascades, we conjectured that there is no linear cascade which is general enough and yet satisfy all above restrictions. Sequential machine synthesis using two dimensional arrays or tree structures is very simple once the difference equations are obtained. The predecessor tree, described in Section 2.5.2, not only helps to find difference equations but is particularly fitted to the tree structure.

The number of cells required for synthesizing any given machine using cellular techniques is also derived. In general, we would try to minimize the memory K for minimal realizations. However, this is not true for all cases.

III. SM SYNTHESIS BY MATRIX

3.1 Introduction

In this chapter, we introduce a completely different approach to the problem of cellular realization of sequential machines. This method is also developed independently by the writer. It was pointed out by R. A. Short* that it has some resemblence to the "Flow Table Logic" introduced by Law and Mealy [15]. However, the differences are significant enough, both in structure and approach, that we present our method here.

3.2 Matrices

First, we introduce the transition matrix. A transition matrix of M is a matrix of size nxn, n being the number of the states of M, which shows the transitive relationship between each pair of states. The entry at the intersection of row s_j and column s_k is $\sum_r i_r$ over all possible r such that $s_k = f(s_j, i_r)$. For example, the transition matrix of M4 is given in Figure 34b. An i_h-transition matrix is a transition matrix for the input i_h. The i_1-transition matrix of M4 is shown in Figure 34c. The i_2-transition matrix of M4 is shown in Figure 34d.

---

* Personal communications.

An output matrix is an nxn matrix, again n being the number of states, whose n diagonal entries are the n states of the machine--the entry at row s_j and column s_j is s_j, etc. All non-diagonal entries are entered with zeros and are of little interest to us. The i_h-output matrix is a diagonal matrix of all states by eliminating the elements corresponding to all zero columns in the i_h-transition matrix. For example, the i_1-output matrix and the i_2-output matrix are the same for M4,

$$\begin{bmatrix} A & O \\ O & B \end{bmatrix}.$$

We define "+" as the normal matrix addition, e.g.,

$$\begin{bmatrix} i_1 & 0 \\ 0 & i_1 \end{bmatrix} + \begin{bmatrix} 0 & i_2 \\ i_2 & 0 \end{bmatrix} = \begin{bmatrix} i_1 & i_2 \\ i_2 & i_1 \end{bmatrix}$$

To synthesize a sequential machine, we first obtain all input-transition matrices and output matrices of the machine. Then, we combine as many input-transition matrices as possible without violating the following restriction.

Restriction: No single entry in a combined matrix can have multiple terms, i.e., no "+" sign can appear in any entry of the matrix.

When some input-transition matrices are combined, the corresponding output matrices are also combined by the rules,

$$A + 0 = A,$$

$$A + A = A,$$

$$0 + 0 = 0.$$

The final synthesis connections are then made according to the final input-transition and output matrices.

3.3 Cellular Synthesis

The matrix synthesis of M4 is shown in Figure 35. The circuit before the final synthesis connections are made consists of

1. p input buses (horizontal lines on top),
2. m(n+q)n cells for an n-state machine with q outputs; m is the final number of input matrices.
3. a horizontal bus line, called H-line, for each row of cells,
4. p vertical lines, called V-lines, for each column of cells, each V-line carries an input signal,
5. two constant buses, carrying 0 and 1, at the bottom.

Algorithm 3.1. (For matrix synthesis of sequential machines)

1. Find all i_h-transition matrices and i_h-output matrices for $h = 1, 2, \cdots, p$.
2. Combine as many i_h-transition matrices as possible without violating the restriction. Call these combined matrices C-matrices. Combine corresponding i_h-output matrices and call them B-matrices. Let m be the number of C-matrices

(hence, also the number of B-matrices).

3. Let $j = 1$

4. For the "matrix" formed by the first n rows and the j-th n columns of cells,

   i. a terminals are connected to appropriate V-lines for proper inputs according to the C_j-matrix,

   ii. b terminals are connected to their respective H-lines,

   iii. the c terminal of each cell is connected to the e terminal of the cell directly above it,

   iv. d terminals are connected to H-lines according to the B_j-matrix.

5. For the (n+1)-st row,

   i. connect all c terminals to the H-line,

   ii. connect each b terminal to the e terminal of the nth row cell directly above it,

   iii. if $g(s_k) = 1$, connect the terminal a of the kth cell to 1-bus; if $g(s_k) = 0$, connect the terminal a of the k-th cell to the 0-bus.

6. Repeat step 5 for (n+2)-nd row, (n+3)-rd row, ···, (n+q)-th row for a machine with q outputs.

7. Is $j = m$? If yes, go to step 9; otherwise go to step 8.

8. Let $j = j+1$, go to step 4.

9. Stop. The output functions can be obtained from the last

q H-lines.

For example, we now synthesize M2 using the matrix structure.

The i_h-transition matrices are:

For i_1,

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} i_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & i_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & i_1 \\ 0 & 0 & i_1 & 0 & 0 \\ i_1 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}.$$

For i_2,

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & i_2 & 0 \\ 0 & 0 & 0 & i_2 & 0 \\ 0 & i_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & i_2 & 0 \\ 0 & i_2 & 0 & 0 & 0 \end{bmatrix} \end{array}.$$

The i_h-output matrices are:

For i_1,

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} A & 0 & 0 & 0 & 0 \\ 0 & B & 0 & 0 & 0 \\ 0 & 0 & C & 0 & 0 \\ 0 & 0 & 0 & D & 0 \\ 0 & 0 & 0 & 0 & E \end{bmatrix} \end{array}.$$

For i_2,

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} A & 0 & 0 & 0 & 0 \\ 0 & B & 0 & 0 & 0 \\ 0 & 0 & C & 0 & 0 \\ 0 & 0 & 0 & D & 0 \\ 0 & 0 & 0 & 0 & E \end{bmatrix} \end{array} .$$

Hence, the C-matrix is

$$C = \begin{bmatrix} i_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & i_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & i_1 \\ 0 & 0 & i_1 & 0 & 0 \\ i_1 & 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & i_2 & 0 \\ 0 & 0 & 0 & i_2 & 0 \\ 0 & i_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & i_2 & 0 \\ 0 & i_2 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} i_1 & 0 & 0 & i_2 & 0 \\ 0 & 0 & i_1 & i_2 & 0 \\ 0 & i_2 & 0 & 0 & i_1 \\ 0 & 0 & i_1 & i_2 & 0 \\ i_1 & i_2 & 0 & 0 & 0 \end{bmatrix},$$

and the B-matrix is

$$B = \begin{bmatrix} A & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & C & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & E \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & B & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & D & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} A & 0 & 0 & 0 & 0 \\ 0 & B & 0 & 0 & 0 \\ 0 & 0 & C & 0 & 0 \\ 0 & 0 & 0 & D & 0 \\ 0 & 0 & 0 & 0 & E \end{bmatrix} .$$

The entire structure is shown in Figure 36.

3.4 Minimal Realizations and Bounds

The number of cells required is

$$N = (n+q) \times n \times m , \qquad (3.1)$$

where n = number of states,

q = number of output functions,

m = number of C-matrices required.

Since n and q are fixed parameters of a given machine, the number of cells required varies with m, the number of C-matrices. For minimal realizations, we want m to be minimal.

Algorithm 3.2 (For finding minimal m)

1. Construct a pxp matrix and let all diagonal entries be 1. (p is the number of input symbols).
2. Compare each of the next-state column with all other next-state columns. If the two columns, say j and k, being compared has no identical next state in every row, enter 1 in the entries of j-th row and k-th column and k-th row and j-th column. Enter 0 for all empty entries of the matrix.
3. By permuting rows and columns (identical permutations must be performed at the same time for both rows and columns. e.g., if row 1 and row 5 are permuted, then columns 1 and 5 must also be permuted), we are able to obtain square submatrices along the diagonal in which each submatrix is a unit matrix (all entries are 1). If t is the minimal number of such unit square matrices required to include every diagonal element, then m = t and is minimal.

Note m is at most n since the matrix has 1 for all diagonal entries to start with.

4. Those rows (or columns) consisting a unit square matrix represent those i_h-transition matrices that can be combined without violating the restriction.

Example: M =

	i_1	i_2	i_3	i_4	Z_1	Z_2
A	B	C	F	A	0	1
B	D	E	D	C	1	0
C	F	G	C	G	0	0
D	C	F	B	F	1	1
E	G	G	F	D	1	0
F	A	B	C	E	0	1
G	E	E	A	B	0	0

The initial 4 x 4 matrix is

$$\begin{array}{c} \\ i_1 \\ i_2 \\ i_3 \\ i_4 \end{array} \begin{array}{c} \begin{array}{cccc} i_1 & i_2 & i_3 & i_4 \end{array} \\ \left[\begin{array}{cccc} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{array}\right] \end{array} .$$

Clearly, three unit square matrices along the diagonal are sufficient to cover all diagonal entries. But if we permute columns 1 and 3 and rows 1 and 3, we come up with the matrix

$$
\begin{array}{c|cc|cc}
 & i_3 & i_2 & i_1 & i_4 \\
\hline
i_3 & 1 & 1 & 0 & 0 \\
i_2 & 1 & 1 & 0 & 0 \\
\hline
i_1 & 0 & 0 & 1 & 1 \\
i_4 & 0 & 0 & 1 & 1
\end{array}
$$

which shows that only two unit matrices are needed. Hence, $n = 2$. 2 is the minimal number because $m = 1$ means the original matrix must be a unit matrix which it obviously is not. The final matrix also shows that i_3-transition matrix and i_2-transition matrix can be combined; also the i_1-transition matrix and the i_4-transition matrix can be combined.

3.5 Summary

The method presented in this chapter was developed directly by utilizing a suitable matrix representation. It is noted that when a machine is in a certain state and is subject to an input, it does two things: it makes a state transition and it produces outputs. If the diagonal elements of an nxn array of cells are thought as representing n states, the transition of states can be accomplished by first moving horizontally and then vertically and the output can be collected by a bottom collection row. The number of cells required for a minimal realization of a given machine is also derived. This method appears to be simpler and more attractive than the first.

IV. INITIALIZATION PROBLEM

4.1 Introduction

Classically, the starting state problem is important because the behavior of the machine is not deterministic unless the starting state is known. More positively, however, we can take the point of view that by starting the machine from different initial states we might be able to specify any one of a diverse set of possible machine behaviors. That is, the specific behavior of the machine may be determined as a function of the initially selected starting state. For example, Figure 37 shows a machine whose behavior as starting from state A is completely different from that of starting from state D.

In sequential circuits different states of the machine are represented by different flip-flop conditions. Generally, we can start the machine from any specified state by setting the flip-flops directly to the conditions corresponding to those of the specified state. But in our cellular realizations of sequential machines, the initialization procedure is not so simple since the states are not represented by flip-flop conditions. If a machine is synthesized in matrix form discussed in Chapter 3, it is still very easy to start it from any desired state. For example, if we wish to start M2 from state A, we simply apply a pulse to the H_A line (see Figure 36) at the same time with this first input. The problem is not so simple with other cellular structures,

however, and we must consider them in a little more detail.

Another way of starting a machine from a specified state is to apply an "initialization input sequence" to force the machine into the desired state. The initial outputs due to the initialization input sequence are discarded. This technique of applying an initialization sequence to force the machine into a certain state can be used to initialize our cellular circuits presented in Chapter 2. But states that can not be reentered present a special problem. These cannot-be-reentered-states, though may not occur very often, do exist and their treatment is the special concern of this chapter.

4.2 Recoverable Machines

A state in a machine is recoverable if the state can be reached from any other states of the machine by a finite sequence of inputs (including the feedback input, if exists). This finite sequence of inputs is called a recovering sequence. There may be more than one such sequence for a given state. We shall always use a minimal recovering sequence, i.e., a recovering sequence of least length. Hence, if we say R is a recovering sequence of state A, it is understood that R is a minimal recovering sequence of state A. A state with no recovering sequence is a transient state. A machine wherein every state is recoverable is called a recoverable machine.

Clearly the problem is caused by those states having no recovering sequences, i.e., transient states. We shall approach this problem of initializing to a transient state by expanding the flow table of M in the expectation that the expanded flow table might make originally transient states recoverable. There are basically two ways that we may consider. We can either enlarge (1) the number of columns (inputs) or (2) the number of rows (states). We shall call the first approach the input approach and the second the state approach. These two approaches are illustrated by state graphs shown in Figure 38.

Figure 38 shows the state graph of M which has three transient states D, E and F. If a third input i_3 is used, transient states D, E and F can be made recoverable as shown in Figure 38b, where transitions indicated by dotted lines are transitions due to the added input i_3. i_3 is used only during the initialization phase. On the other hand, Figure 38c shows the recoverable version of the machine using an added state G. i_x is a special signal used during the initialization phase. Note that the only way for states A, B, or C to reach any added states (for that matter, even states D, E, F) is by signals other than i_1 and i_2. This signal, however, could be the one that indicates the initialization phase. Also, note that either i_1 or i_2, if combined with the initialization signal, can be considered as a third signal. Thus, whether we use the input approach or the state approach, one thing is common, i.e., a signal to indicate that the machine is in the

initialization phase is required. Since we shall show in this chapter that when input approach is used, a nonrecoverable machine can be made recoverable by using just one additional signal (and this can be just the signal that indicates the initialization phase), we will not discuss the state approach that requires additional states on top of the initialization signal.

4.3 Input Approach

One basic idea behind this method is that if the i_h-column in the flow table is a reset column for state s_j, we can always get to state s_j by simply applying the input i_h.

Theorem 4.1. A nonrecoverable n-state machine can be made recoverable by adding at most n columns.

Proof: 1. Add a reset column for each transient state. This makes all transient states recoverable.

2. Since there are at most n transient states, we need to add at most n reset columns.

3. We have made all states recoverable, hence a recoverable machine. Q.E.D.

Usually, a machine has only a few transient states, hence the number of columns to be added is usually not many. Example: Figure 39a shows a machine M5 which is not recoverable since states B

and E are not recoverable. Two next-state columns are added to M5 in Figure 39b and the machine is now recoverable.

Note two things, however.

1. Once state B is made recoverable, E also becomes recoverable. So i_4 is not essential.
2. The i_3 column does not have to be a reset column. Since states A, C and D are recoverable, we only require that state B be the next state of one of these three states. This will make state B recoverable. Same argument applies to state E. If we choose states B and E to be the next states of different recoverable states, say states A and C respectively for this example, then only one additional column is needed. The rest of the entries in that column are immaterial for the time being.

This brings us another idea behind this method, that is, if a column of a flow table M is a complete permutation column, then M is recoverable. By a complete permutation column we mean a permutation column from which a single-loop state graph that includes all states of M can be constructed.

Note that not all permutation columns are complete permutation columns. For example, Figure 40 shows a permutation column but not a complete permutation column. Figure 41 shows a complete permutation column. It is very easy and straight forward to form a complete

permutation column. One way to construct such a column is by a sort of end-around shift of the present state column. Figure 41 is one example. Thus, we have proved the following theorem.

Theorem 4.2. All nonrecoverable machine can be made recoverable by adding at most one column.

Obviously, the column added is not unique since a complete permutation column can be constructed by other methods. Note in our cellular realizations, the number of cells required is proportional to the number of input variables. Since the inputs are normally of binary signals, 5 columns and 8 columns make no difference in hardware since they both require 3 input variables. Hence, if we need at least 5 columns, then we may as well expand it to 8 columns if we can gain something. This something gained turns out to be shorter recovering sequences.

Let R = a set of recoverable states.

T = a set of transient states.

u = the length of a longest recovering sequence of a recoverable machine,

v = number of added input variables.

From Theorem 4.2 v is at most one.

We define a minimal recoverable machine M as a machine with minimal u with the prerequisite that v is minimal. Since v can be

easily found to be either 0 or 1, the problem is to keep u minimal. This is a very complicated and involved problem. There seems no way to find the minimal recoverable machine. We can only present an algorithm which tends to give a good low u value. First, we need some definitions.

A state in R (set of recoverable states) which has a shortest recovering sequence among all states in R is called an S-state. A state in R which has a longest recovering sequence among all states in R is called an L-state. A partial state graph which is formed by transient states only is called a T-state graph. A state s which has the most number of distinct successor states is a preferred state. Notation #(P) represents the number of elements in the set P.

Algorithm 4.1. (For constructing a minimal recoverable machine)

1. If M is recoverable, go to step 11; otherwise go to step 2.
2. Find R_0 and T_0 of M.
3. Construct a T-state graph.
4. Add a new column. Let $j = 0$.
5. If $\#(R_0) = 0$, choose a preferred state from the T-graph and make this a reset column for the chosen state, go to step 2. If $\#(R_0) \neq 0$, go to step 6.
6. Select an S-state r_j from R_j; select a preferred state t_j from the T-state graph formed by T_j; enter t_j as the next

state of r_j in the new column.

7. Find $T_{j+1} = T_j - t_j$ - (all successors of t_j).

8. If $\#(T_{j+1}) = 0$, go to step 11; otherwise go to step 9.

9. Find $R_{j+1} = R_j + t_j - r_j +$ (all successors of t_j).

10. $j = j+1$, go to step 6.

11. If the number of total columns $= 2^{p+v}$, go to step 20; otherwise go to step 12. (p is the original number of input variables and v is the added number of input variable, which is either 0 or 1.)

12. Add a new column. Let $j = 0$.

13. R_0 should be the set of all states now.

14. Select an S-state r_j from R_j; select an L-state p_j from R_j.

15. If the recovering sequence of p_j is two or more greater than that of r_j, go to step 17; otherwise go to step 16.

16. If $j = 0$, and if this is not the last column to be added, select a preferred stated from R_0 and make this column a reset column for that state and go to step 12. If $j = 0$ and this is the last column to be added, make this a reset column for p_j, and go to step 20. If $j \neq 0$, go to step 11.

17. Enter p_j as the next state of r_j and erase all p_j in other entries of added columns.

18. $R_{j+1} = R_j - r_j + p_j$.

19. j = j+1, go to step 14.

20. The unspecified entries of added columns are don't cares. If an added column contains only one state, make that column a reset column for that state.

21. Stop.

Again, the expanded machine is not necessarily unique because of the choices of states in many steps are not necessarily unique. Since most practical machines have relatively few degenerate states, a machine with the minimal u can usually be obtained with more experience. Many optimal solutions can actually be obtained by inspection. An example is given in Figure 42 where X is the added input and can be considered as the signal for initialization.

4.4 Implementation of Recoverable Machines

Now that we have made all machines recoverable, how do we implement them? The answer is simply by following the procedure described in the earlier chapters. We illustrate this by an example. Figure 43 shows a machine, M6, and the successive steps in synthesizing it directly in the nonrecoverable version. Figure 44 shows the same machine, M6, which is first made recoverable and then synthesized by following the same procedure.

Now we compare Figure 43g with Figure 44g and Figure 43h with Figure 44h. M6 can be identified from M6' as indicated by

dotted lines. In the tree realizations, we use Figure 44g during the initialization phase and change the bottom half boundary conditions to 0's during the normal machine operation. In the array realizations, we simply let $X = 0$ at all times when we are not initializing. To initialize a machine to a desired state is to apply the recovering sequence of that state to the machine.

4.5 Initialization Sequence

The tree structure of Figure 45 is called a recovering tree. It is a slightly modified version of the D-successor tree. The only difference between the two is their terminating conditions. The terminating condition for a recovering tree are:

1. When every state of the machine appears at least once as a singleton in the tree, terminate the entire tree.
2. When a state set repeats a state set appeared at the same or earlier level, terminate that branch.
3. When the state set is empty, terminate that branch.

When all branches are terminated, the tree is terminated. The sequence of branches that take the zeroth level state set to a singleton state s_j are recovering sequences for state s_j. We always choose a minimal sequence as the recovering sequence. A recoverable machine should have its recovering tree terminated by condition 1. From Figure 45 we can see that the recovering sequences for the four states

of M6' are

for A------$X\bar{Y}$,

B------$X\bar{Y}$, $\bar{X}\bar{Y}$,

C------XY,

D------$X\bar{Y}$, $\bar{X}Y$.

The following two theorems are obvious.

Theorem 4.3. A recoverable machine is feedback free only if its nonrecoverable version is feedback free.

Theorem 4.4. A recoverable machine is FIFM only if its nonrecoverable version is FIFM.

4.6 Summary

There are states in a machine that are not attainable by applying a finite sequence of inputs. If we wish to start the machine from any of these states, we must first make the machine recoverable. Methods were developed to make nonrecoverable machines recoverable by adding at most one input variable to the machine. This input is the signal used to indicate that the machine is in the initialization phase. The synthesis procedures of recoverable machines are identical to those given in earlier chapters. The circuit for the original nonrecoverable machine is a distinct section of the circuit for the new recoverable machine. The recovering sequence for each state can easily be obtained

by constructing a recovering tree. Since most practical nonrecoverable machines have none or only a few transient states, the process is usually quite simple.

V. RELIABILITY AND PROGRAMMABILITY

5.1 Introduction

It is safe to predict that future computing system will continue to increase the demands on several sophisticated design areas. They will need to be more readily expandable and modifiable. Also automatic error detection and correction will play a more significant role. Therefore, modularity, reliability and programmability are important aspects of a digital system. The past chapters can be considered as an effort to make hardware modular and readily expandable. We now examine the reliability and programmability characteristics of our cellular sequential machines.

As the digital system becomes more complex, it demands components of unrealistically high reliability in order to yield a reliable system. In recognition of the impossibility of building perfect components, significant effort has been devoted to the development of techniques for realizing reliable digital systems from nonperfect components in recent years. Thus, in addition to making all components as reliable as possible and to utilizing proper worst case design and adequate assembly technology, it is also necessary to incorporate into the system some kind of redundancy so that the desired reliability of overall system operation may be achieved.

There are two sources of error in a digital system. One is the transient error due to time delay or noise pickup, etc. The other is the permanent error due to the failure of a component or components. We shall consider primarily the permanent type of error since transient errors can generally be minimized by careful design and choice of environment. In electronic gate network terms, therefore, interest is in the correction of faults caused by inputs or outputs of a logic block to be either "stuck at one" or "stuck at zero", though some of the techniques we shall discuss will correct transient errors as well.

Redundancy techniques employed for improving reliability of a digital system are generally classified into two categories: static redundancy and dynamic redundancy (see [21] for an excellent survey and discussion). Techniques presented in this chapter will, strictly speaking, correct or locate only a single fault. This is based on the assumption that the probability of simultaneous occurance of two or more faults is very small. Of course, these techniques can be modified to include multiple faults. Also, we shall use different cellular structures to illustrate each redundancy technique. It must be remembered that these techniques are perfectly general and can be applied to any other cellular structures.

As the microelectronic technology advances, multipurpose devices begin to appear. Programmable circuits represent one step further in this direction. A programmable circuit is a circuit that is

able to perform many different system functions (in our case, the realization of many different sequential machines) by proper reprogramming. One of the advantages of our cellular circuits is that they can be made readily programmable. Of course any programmable circuit has its special limitations and these might include, for example, the number of inputs, the number of outputs and/or the number of states. The limitations for our cellular sequential circuits are slightly different according to the particular implementation methods used, hence will be presented separately for each case. The programming techniques we shall develop in this chapter are divided into two categories: minterm select, and coincidence or linear select. They are discussed in Sections 5.3.1 and 5.3.2.

5.2 Reliability Improvements

5.2.1 Static Redundancy

Static redundancy (or active redundancy) requires two or more identical units operating at same time. In the event of failure of one or more, the also-operating-but-nonfaulty unit or units will continue to provide adequate responses. The most important technique in this category at the network level is fault-masking by replication. The faults are corrected within the network and terminal behaviors are not affected until the tolerance limit of the network is exceeded. Depending

upon the network structure, static redundancy techniques are usually divided into two groups: voting schemes and nonvoting schemes. In both cases, the correction of errors is accomplished "on-line" while performing digital logic.

A. Voting Scheme. In this scheme, each circuit is replicated several times, each independently of the other, and then a weighted measure, usually a majority vote, is taken over these outputs. The weight placed on each line may be fixed or variable. Figure 46 illustrates a triplicated version of the cell used in matrix structures. For this enlarged cell to operate successfully, at least two of the three parallel units must have correct signals. The interconnections between cells are also triplicated such that each unit in a cell receives an input from a unit in a preceeding cell. Any single fault in the circuit will be corrected after it goes through a majority gate. This scheme will also correct multiple faults if they do not occur at the same level, i.e., if each majority gate has at most one faulty input.

B. Nonvoting Scheme. In this scheme, we replace each component by a network of components. Faulty signals are corrected down the path by mixing with correct signals. The only difference between a nonvoting scheme and a voting scheme is in the absence or presence of a decision making (vote taking) element. The most famous in this class of redundancy technique is Tryon's "quadded logic". Figure 47 illustrates such a version for the cell used in fixed function

array. The 16 NAND-gate network shown in Figure 47 replaces each 4 NAND-gate group in the original array. The interconnections for each NAND-gate are such that the output interconnections are different from that of the input. For example, the inputs for the NAND-gate A is from the first and fourth line while the output of A goes to the first and the second line. The error correcting mechanism is explained in Figure 48. The quadded logic has the ability to correct any single fault and many more as long as 1. faulty gates are not directly connected to each other, and 2. each gate has at most one faulty input.

5.2.2 Dynamic Redundancy

Dynamic redundancy (or standby redundancy) requires schemes for the detection and diagnosis of faults and replacement of faulty parts by standby units either automatically or manually. To perform correction operation, normal operation of the system is usually interrupted. Most of the studies done in this area are in the systems or the subsystems level. We shall first illustrate the repairing process and then the development of a diagnostic software.

Assume that a cut-point cellular array is aranged as shown in Figure 49 where spare columns are normally grounded at the bottom. Also for simplicity, all cells below and including the collection array are cut to perform the OR operation. If a faulty element is discovered

in the main array, say row i and column j, the fault is corrected by following these steps:

1. Gound column j at the bottom so that this column is, in effect, eliminated from the collection array.
2. Choose a spare column, u, and cut the cells to match those of column j.
3. Remove the grounding of column u.

If a faulty element exists in the collection array, say row k and column p, the fault is corrected by:

1. Use a spare row v to replace row k.
2. Obtain the output from row v instead of row k.

Note that a faulty element in either spare rows or columns does not affect the output.

The detection and location of faults, however, represents a major problem. The method presented below is good for any single error detection and location. For multiple errors, similar steps may be followed but becomes much more involved. We use the simple tree structure of Figure 50 for clearer illustrative purposes. Similar procedures can be used for other type of cellular structures.

In general, if

1. a cell has 1 output at all times, then we don't care if the cell is stuck at 1;

2. a cell has 0 output at all times, then we don't care if the cell is stuck at 0;
3. two or more errors are such that they cancel each other and the outputs are not affected, we don't care.

In other words, we are only interested in obtaining the correct output response. The output function of the tree structure shown in Figure 50 is given, in terms of a truth table, in Figure 51a. If cell A is a faulty cell, i.e., it is stuck at 1 (we don't care if it is stuck at 0), the output would be as indicated by f_A in Figure 51b. If cell B is a bad cell, the output would be given by f_B. If cell C is bad, we must consider two cases: that of stuck at 1 and that of stuck at 0. We use f_{C1} for cell C stuck at 1 and f_{C0} for cell C stuck at 0. Other functions in Figure 51b are similarly constructed. Note the regular pattern in these functions. For example, f is the same as the boundary conditions; f_A is the same as f except the first two entries are changed to 1's (due to cell A stuck at 1); f_B is the same as f except that the 3rd and the 4th entries are changed to 0's (due to cell B stuck at 0). We then rearrange the rows of Figure 51b to fit a testing sequence of 00011101000. This is shown in Figure 51c. Thus, if an input sequence of 00011101000 is applied and the output sequence is 001001100, we conclude that cell D is stuck at 0. In cases where feedback exists, the feedback path is broken and the above procedures

are followed. This is one of the reasons that we consider only a single feedback.

5.3 Programmability

5.3.1 Minterm Select

Recall that the boundary conditions in the tree structure correspond to minterms. Since a sequential machine can be expressed by sum of appropriate minterms, different sequential machines are realized by applying different boundary conditions. These boundary conditions can be obtained from a register which in turn can be controlled by programming. This very same technique can also be applied to array structures. For example, Figure 52 shows a cut-point cellular array from which each column of cells realizes one minterm. Thus, for a sequential machine with m variables (remember that different time subscripts represent different variables) we need 2^m columns of cell. The contents in the register just above the bottom collection now decide which minterm or minterms are to be included, with the help of AND gates, for implementing a particular machine.

The bounds on the class of sequential machines that can be realized are p, q and K. In other words, a programmable cellular network which is originally designed to realize a sequential machine with p total inputs, q total outputs and memory K, it can be

reprogrammed to implement any sequential machine which satisfies the following:

total number of inputs $\leq$ p,

total number of outputs $\leq$ q, and

memory $\leq$ K.

5.3.2 Coincidence or Linear Select

Coincidence and linear select methods are well known in the addressing systems of core memories. These techniques can be used to make our matrix array programmable and are described below.

A. Coincidence Select. Consider the array in Figure 53a and the cell structure in Figure 53b, each cell is "addressed" by a horizontal addressing line and a vertical addressing line. The intersect of a vertical addressing signal and a horizontal addressing signal is fed into a shift register. The shift register stores the program from which the operation of the cell is determined. It is obvious that one row of cells can be programmed at the same time. Whether the signal to be stored in the register is 0 or 1 is determined by the signal from the vertical addressing line. For example, if row 2 is to be programmed, 1 is applied to line PH2 and either 0 or 1 to PV lines depending on the functions C_{21} and C_{22} are to perform. For the cell structure shown, three successive programming signals are needed for each cell. The first one (stored in R_1) determines

whether the delayed cell output is to be fed to the horizontal bus line. The second signal (stored in R_2) determines whether input x_2 is an input to the cell or not. The third signal (stored in R_3) determines whether x_1 is an input to the cell.

B. Linear Select. A cell structure for this scheme is shown in Figure 54. In linear select, the programming signals are entered through a single programming bus for all cells. There are many ways this programming bus can be arranged. Some of these are given in Figure 55. In whatever form the programming bus is arranged, the registers of all cells are connected in series. The earlier programming signals are propagated into registers further down the line. Thus, a matrix with m cells and each cell with an n bit register needs mn programming signals. Cell operations are identical to those of coincidence select and are determined by the contents of the register.

In both coincidence select and linear select, a matrix designed for n state sequential machine with p inputs and q outputs can realize any n or less state sequential machine with p inputs and q outputs. Note that for programmable matrix structure, $m = 1$ (see p. 50) and the restriction given on page 49 is no longer necessary.

5.4 Summary

The need for redundancy to increase the overall system reliability is apparent. Examples of both static redundancy and dynamic redundancy techniques have been studied. In static redundancy, errors are corrected "on-line" while the circuit is performing digital logic. In dynamic redundancy, diagnostics are used to locate the fault and spares are switched in to replace faulty elements. For a system employing static redundancy, the amount of hardware components are usually many times larger. On the other hand, dynamic redundancy techniques may not require as many hardware components but generally require machine down time for diagnostics and switchover. Also the task of developing a good diagnostic test can be quite involved. From the LSI point of view, static redundancy techniques seem to be favorable since extra components add little to the cost (if the limitation is not exceeded). Also note that reliability is already increased by LSI because of fewer solder joints and contacts.

The programmability feature is one advantage in using cellular circuits. By programmability we mean the ability of a circuit to perform various functions by reprogramming. Two techniques are presented: A. minterm select and B. coincidence or linear select. Since any sequential function can be expressed as combinational like equations, these equations can be expanded as the sum of appropriate

minterms. Different functions are realized by collecting different minterms. Coincidence select and linear select techniques are borrowed from conventional core memory addressing systems.

VI. SUMMARY AND CONCLUSIONS

6.1 Summary

With the advancing solid-state technology, it is necessary to develop new techniques for synthesizing digital networks. The regular pattern of cellular circuits seems to be the best fitted for the new LSI technology. Recently, cellular implementations of combinational circuits have received considerable attention but very little attention has been given to sequential circuits. In this paper, we have presented two new methods for realizing sequential machines, both using cellular circuits. Moore machines are assumed.

The first method converts sequential functions into combinational like equations. In order to do so, the machine must be either definite or FIFM. Successor trees and pair tables are presented as the means for testing whether the given machine satisfies this requirement. If the machine is neither definite nor FIFM, it is made FIFM by constructing a proper feedback function. When a machine is definite or FIFM, it can be described by either predecessor trees or difference equations. These equations are combinational like equations and can easily be implemented by conventional combinational cellular circuits together with delay elements.

The second method is developed more or less by direct application of a convenient matrix representation. It is noted that when a machine is in a certain state and is subject to an input, it does two things: it makes a state transition and it produces outputs. If the diagonal elements of an nxn array of cells are thought as representing n states, the transition of states can be accomplished by first moving horizontally and then vertically and the output can be collected by an added bottom collection row. A formal synthesis method was then developed using these matrix type cellular arrays. This method appears to be simpler and more attractive then the first.

In both cases, bounds on the number of cells were established and minimal realizations were studied. Methods for starting these cellular machines were also investigated. In order to make the machine more flexible, techniques were devised to initialize the machine into any state desired.

An important engineering problem, reliability, was also considered. Although reliability in general is increased by using LSI techniques because of fewer solder joints and contacts, there are cases where ultra reliable systems are desired. Various existing redundancy techniques are all suitable for reliability improvement of our cellular circuits. However, static redundancy techniques seem to be the better solution for LSI cellular circuits. One of the advantages of these cellular circuits is the ease by which they can be made

programmable. That is, our cellular circuits are easily modified so that it is able to implement different sequential machines, within a given class of sequential machines, by programming. The minterm select (by collecting proper minterms) technique was introduced to make the first kind of cellular circuits programmable, while the second type of cellular circuit relied on the coincidence or linear select method used in conventional core memory addressing systems.

6.2 Problems for Further Research

There are several problems in the course of this study that are not fully resolved. They are given below.

1. Is there an easier way of constructing the feedback function if the machine is not FIFM to start with?
2. Is the conjecture given on page 38 true? Or, can we find a simple sequential machine (which can be considered as a single cell) such that all sequential machines can be realized by cascading several of these simple sequential machines?
3. What if multiple feedback is used? What are some of the tradeoffs? Even the tradeoff problem among parameters p, q and K for a single feedback are not fully known for the array structure.
4. Algorithm 4.1 does not guarantee a minimal recoverable machine. Can a better but not too complicated (for example,

by exhaustive testing) algorithm be developed?

5. Can cellular structures be modified so that for a particular circuit, a large class of sequential machines can be realized by reprogramming? In other words, can we find a way to ease our limitations on the class of sequential machines that is realizable by reprogramming?

6.3 Conclusions

The purpose of this study has been to develop cellular methods of synthesizing sequential machines. From the point of view of LSI, cellularization represents a great saving in cost, space and weight. From the designer's point of view, cellularization does away with the difficult state assignment problems (although other problem may be introduced). A further advantage of cellularization lies in its programmability. Future computing systems will be large and complex, and must be easily modifiable. Programmable sequential machines are thus important in such reconfigurable systems. Also static redundancy techniques can be employed in LSI without amplifying the cost. Further, because of the inherent reliability of LSI circuits, very reliable sequential machines can be designed.

The techniques presented in this paper, of course, only open up new approaches to the cellular design problem. Although methods developed are not exploratory, very good insight into the problem has

been gained. It is anticipated that further improvements and improved design techniques will result in the near future.

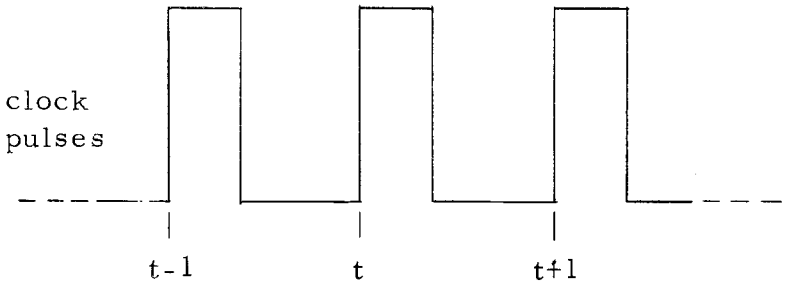


Figure 1. Time unit of a synchronous sequential machine.

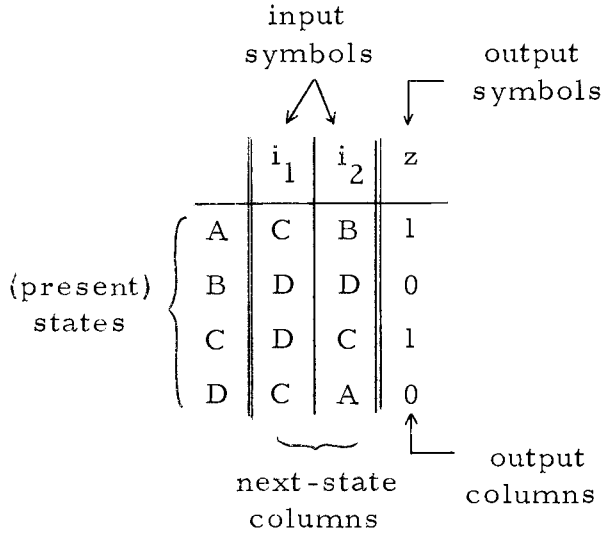
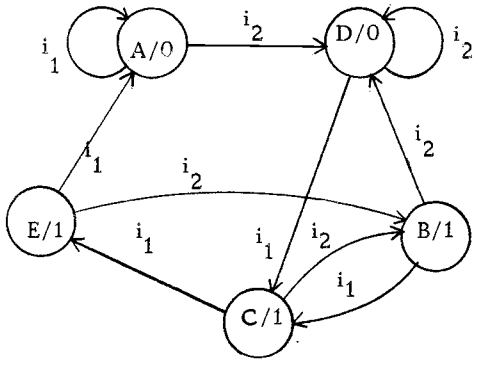


Figure 2. Flow table representation of a Moore machine, M1.

	i_1	i_2	z
A	A	D	0
B	C	D	1
C	E	B	1
D	C	D	0
E	A	B	1

(a)



(b)

Figure 3. (a) Flow table of M2.
(b) State graph of M2.

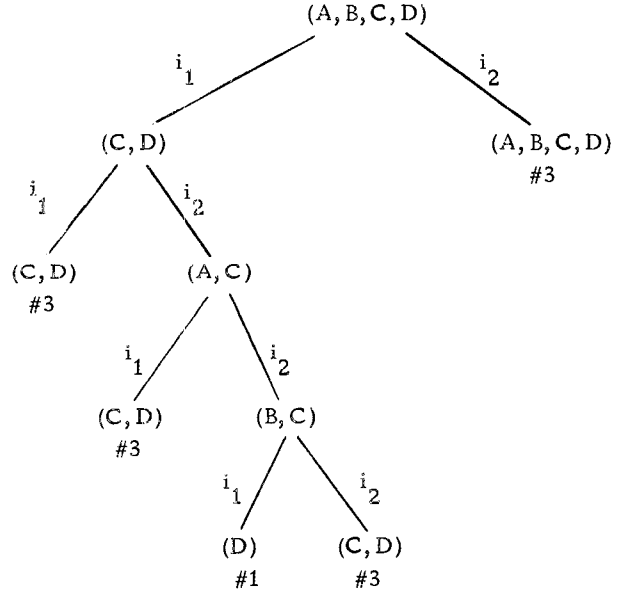


Figure 4. D-successor tree test of M1.

	i_1	i_2
AB	CD	BD
AC	CD	BC
AD	C	AB
BC	D	CD
BD	CD	AD
CD	CD	AC

Figure 5. D-pair table test of M1.

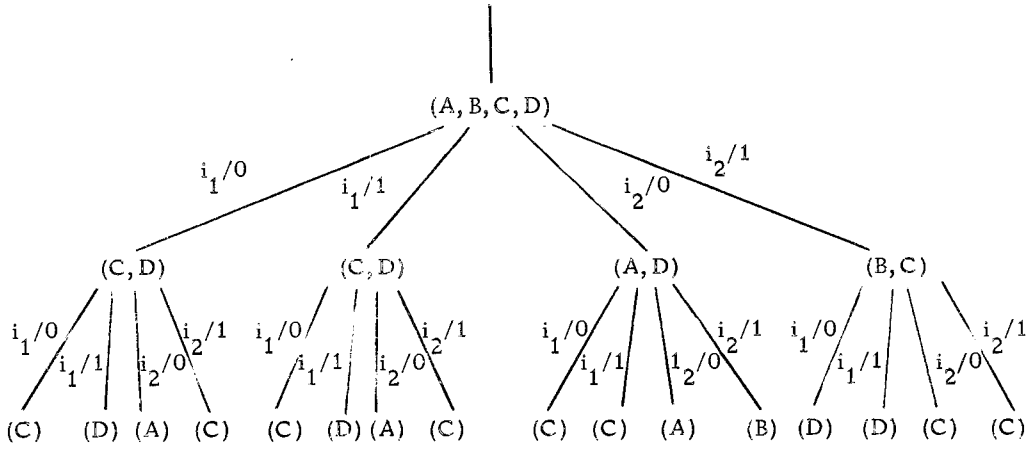


Figure 6. F-successor tree test of M1.

k		i_1	i_2
0	~~AB~~	—	—
1	~~AC~~	~~CD~~	~~BC~~
0	~~AD~~	—	—
0	~~BC~~	—	—
1	~~BD~~	~~CD~~	~~AD~~
0	~~CD~~	—	—

Table 7. F-pair table test of M1.

	i_1	i_2	z
A	C	B	0
B	D	D	0
C	D	C	1
D	C	A	1

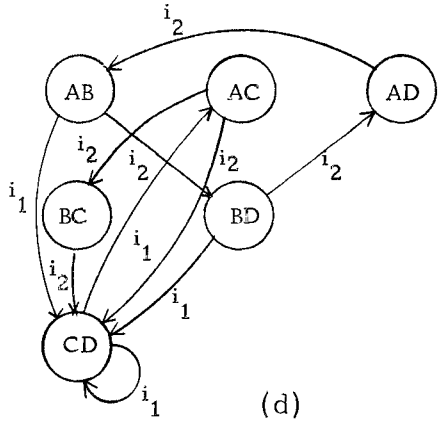
(a)

	i_1	i_2
AB	CD	BD
AC	CD	BC
AD	C	AB
BC	D	CD
BD	CD	AD
CD	CD	AC

(b)

k		i_1	i_2
	AB	CD	BD
0	~~AC~~	—	—
0	~~AD~~	—	—
0	~~BC~~	—	—
0	~~BD~~	—	—
	CD	CD	AC

(c)



(d)

Figure 8. (a) Flow table of M3.
(b) D-pair table test of M3.
(c) F-pair table test of M3.
(d) D-implication graph of M3.

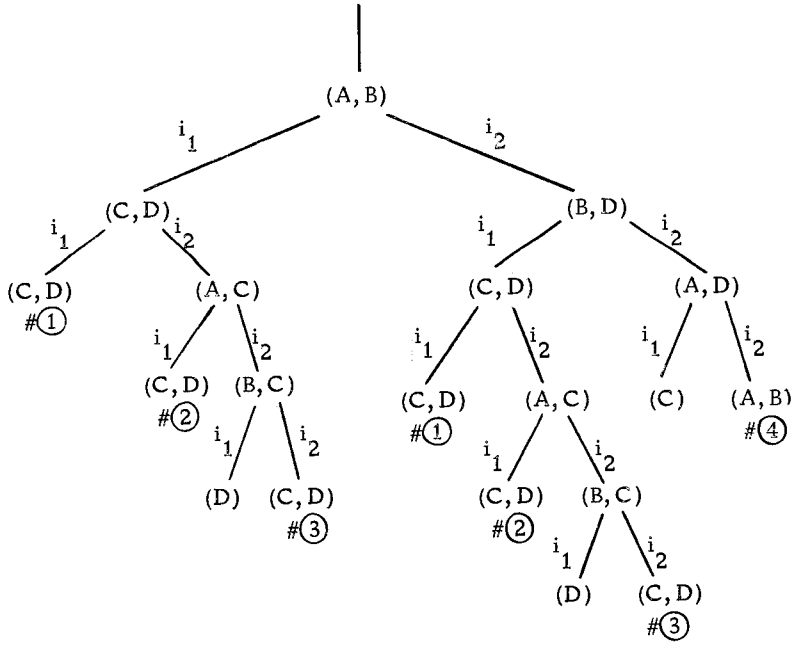
	i_1	i_2	Z	F
A	C	B	0	1
B	D	D	0	0
C	D	C	1	1
D	C	A	1	0

(a)

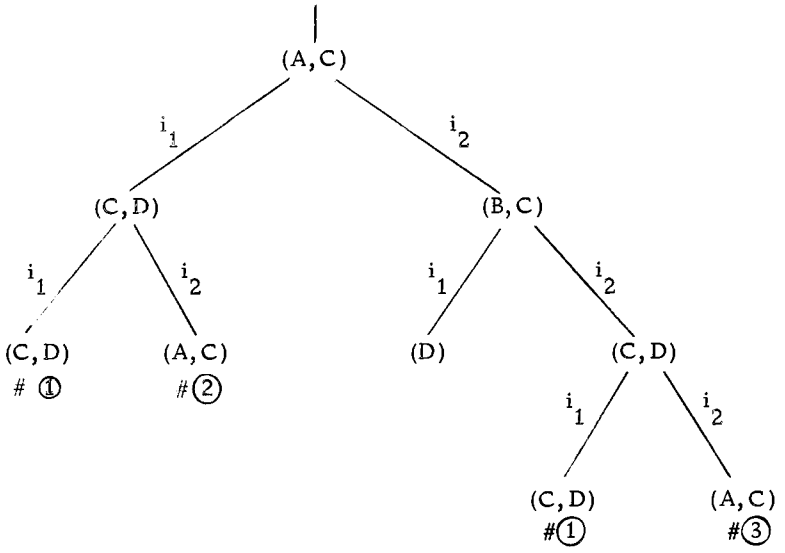
k		i_1	i_2
0	~~AB~~	~~CD~~	~~BC~~
1	~~AC~~	~~CD~~	~~BC~~
0	~~AD~~	—	—
1	~~BC~~	—	—
1	~~BD~~	~~CD~~	~~AD~~
0	~~CD~~	—	—

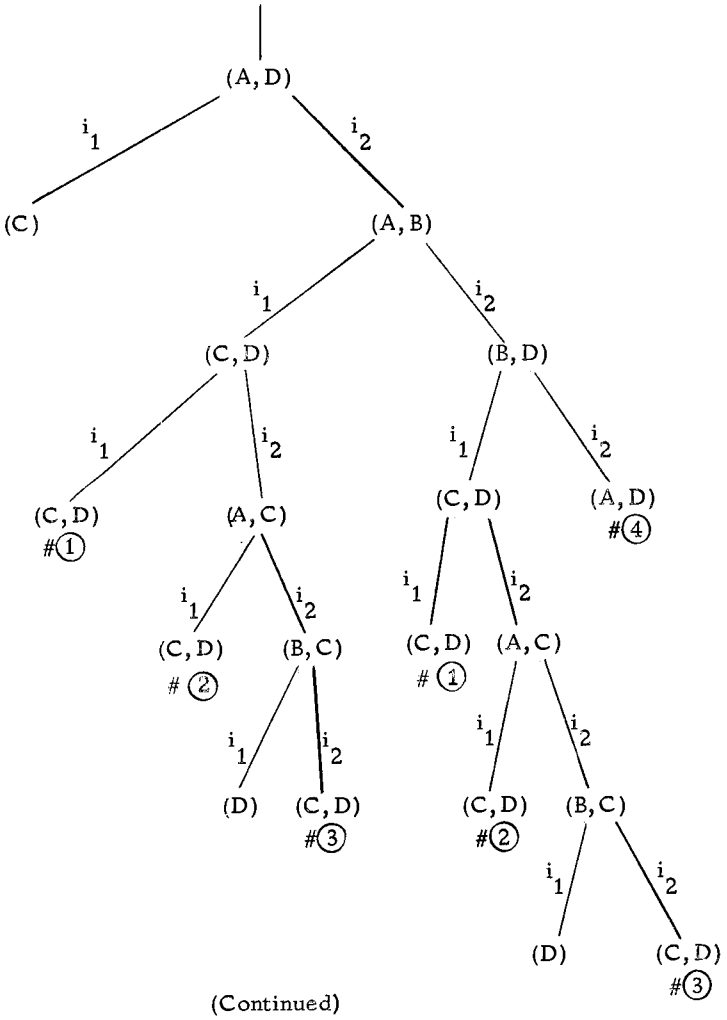
(b)

Figure 9. (a) Flow table of M3'--M3 with an added output, F.
(b) F-pair table test of M3'.

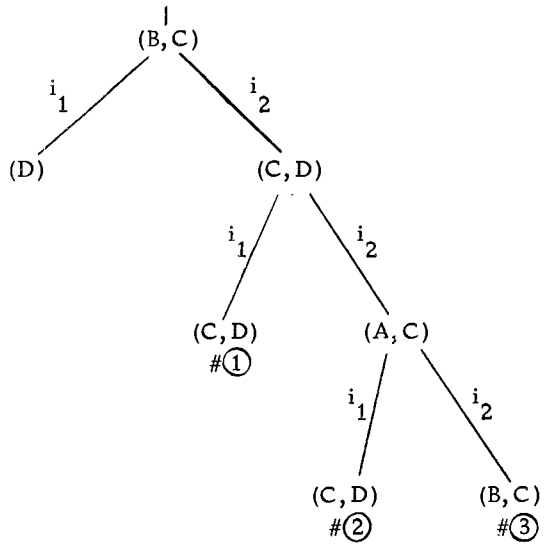


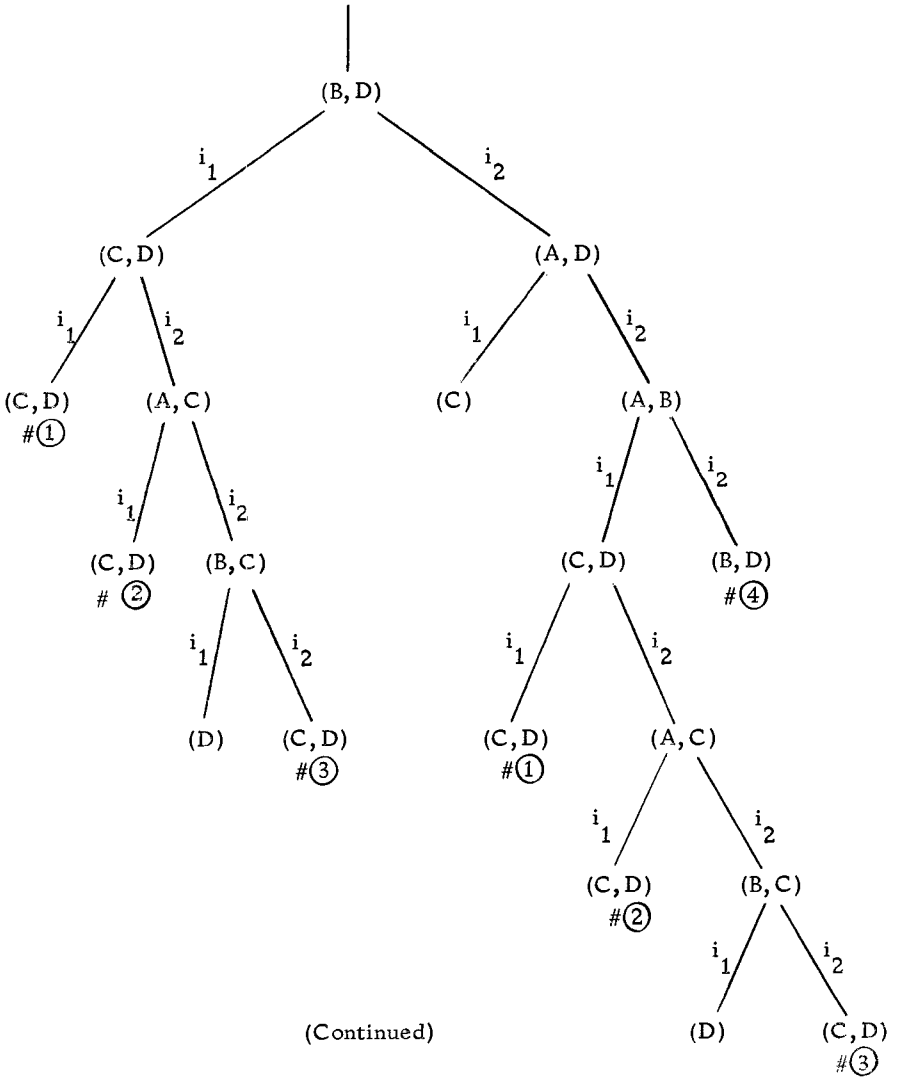
(Continued)


(A, C)
i1
i2
(C, D)
(B, C)
i1
i2
i1
i2
(C, D)
①
(A, C)
#②
(D)
(C, D)
i1
i2
(C, D)
#①
(A, C)
#③


(A, D)
i1
i2
(C)
(A, B)
i1
i2
(C, D)
(B, D)
i1
i2
i1
i2
(C, D)
#①
(A, C)
(C, D)
(A, D)
#④
i1
i2
i1
i2
(C, D)
#②
(B, C)
(C, D)
#①
(A, C)
i1
i2
i1
i2
(D)
(C, D)
#③
(C, D)
#②
(B, C)
i1
i2
(D)
(C, D)
#③

(Continued)


(B, C)
i1
i2
(D)
(C, D)
i1
i2
(C, D)
#①
(A, C)
i1
i2
(C, D)
#②
(B, C)
#③


(B, D)
i1
i2
(C, D)
(A, D)
i1
i2
i1
i2
(C, D)
#①
(A, C)
(C)
(A, B)
i1
i2
i1
i2
(C, D)
#②
(B, C)
(C, D)
(B, D)
#④
i1
i2
i1
i2
(D)
(C, D)
#③
(C, D)
#①
(A, C)
i1
i2
(C, D)
#②
(B, C)
i1
i2
(D)
(C, D)
#③

(Continued)

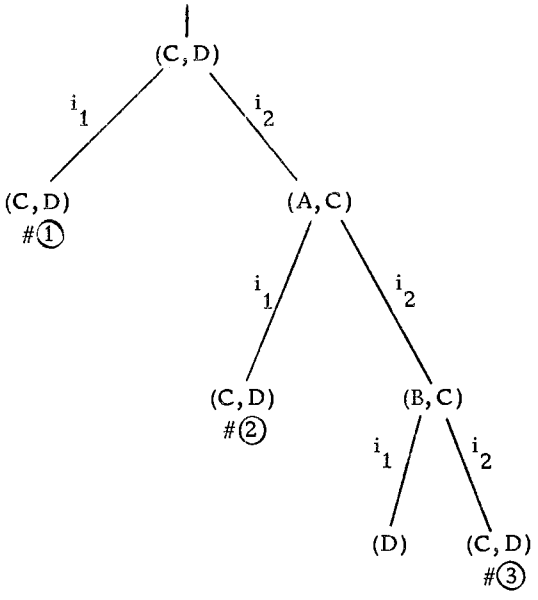


Figure 10. D-pair trees of M3 (# indicates branch that is terminated by condition 3. Numbers inside small circles are elementary loop numbers as given on page 20.)

	$\overline{X}$	X	Z
A	A	D	0
B	C	D	1
C	E	B	1
D	C	D	0
E	A	B	1

(a)

k		$\overline{X}$	X
3	~~AB~~	~~AC~~	~~D~~
2	~~AC~~	~~AE~~	~~BD~~
3	~~AD~~	~~AC~~	~~D~~
1	~~AE~~	~~A~~	~~BD~~
3	~~BC~~	~~CE~~	~~BD~~
0	~~BD~~	~~C~~	~~D~~
3	~~BE~~	~~AC~~	~~BD~~
3	~~CD~~	~~CE~~	~~BD~~
2	~~CE~~	~~AE~~	~~B~~
3	~~DE~~	~~AC~~	~~BD~~

(b)

Figure 11. (a) Flow table of M2 using X as the input variable.
(b) D-pair table test of M2.

	$\bar{X}$	X	Z
A	C	B	1
B	D	D	0
C	D	C	1
D	C	A	0

(a)

	$\bar{X}\bar{F}$	$\bar{X}F$	$X\bar{F}$	XF	Z
A	—	C	—	B	1
B	D	—	D	—	0
C	—	D	—	C	1
D	C	—	A	—	0

(b)

Figure 12. (a) Flow table of M1 using X as the input variable.
(b) Expanded flow table of M1 to include the feedback input.

	$\bar{X}$	X	Z	F
A	C	B	0	1
B	D	D	0	0
C	D	C	1	1
D	C	A	1	0

(a)

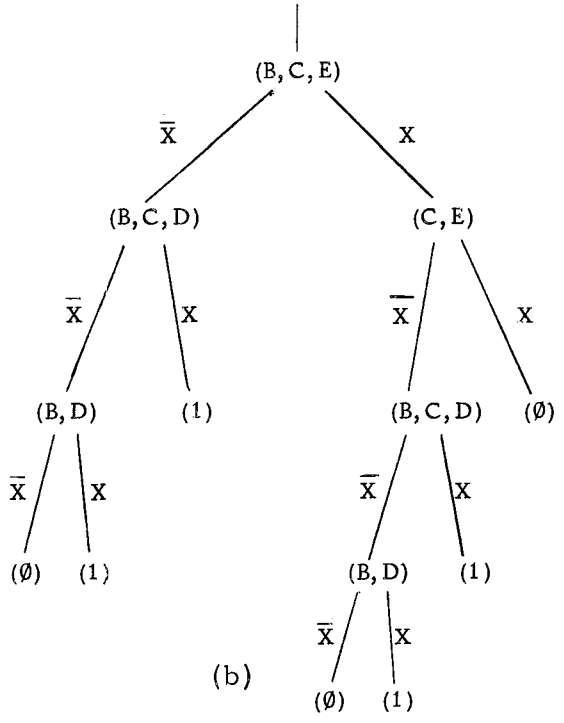
	$\bar{X}\bar{F}$	$\bar{X}F$	$X\bar{F}$	XF	Z	F
A	—	C	—	B	0	1
B	D	—	D	—	0	0
C	—	D	—	C	1	1
D	C	—	A	—	1	0

(b)

Figure 13. (a) Flow table of M3'.
(b) Expanded flow table of M3' to include the feedback input.

	$\bar{X}$	X	Z
A	A	D	0
B	C	D	1
C	E	B	1
D	C	D	0
E	A	B	1

(a)

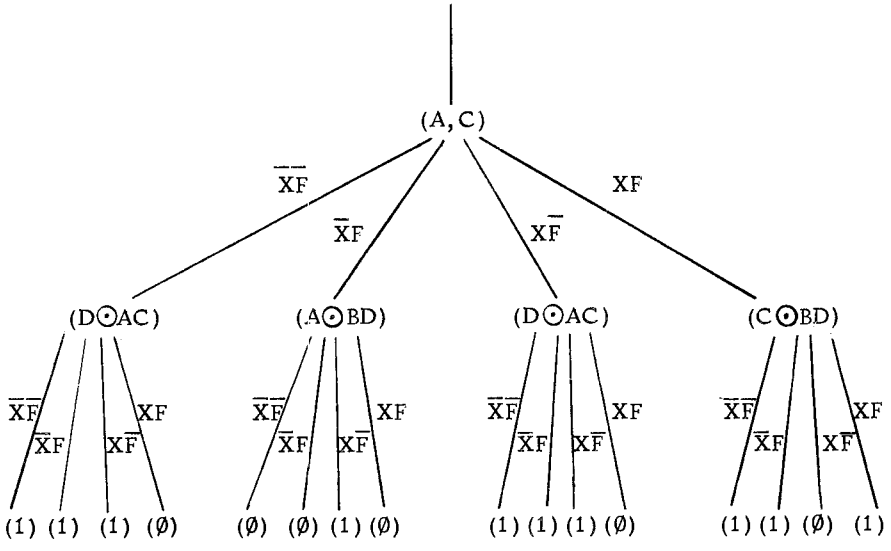


(b)

Figure 14. (a) Flow table of M2.
(b) Predecessor tree of M2.

	$\bar{X}\bar{F}$	$\bar{X}F$	$X\bar{F}$	XF	Z
A	—	C	—	B	1
B	D	—	D	—	0
C	—	D	—	C	1
D	C	—	A	—	0

(a)

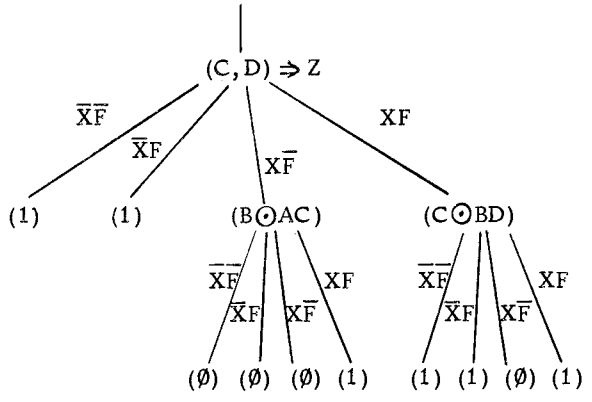


(b)

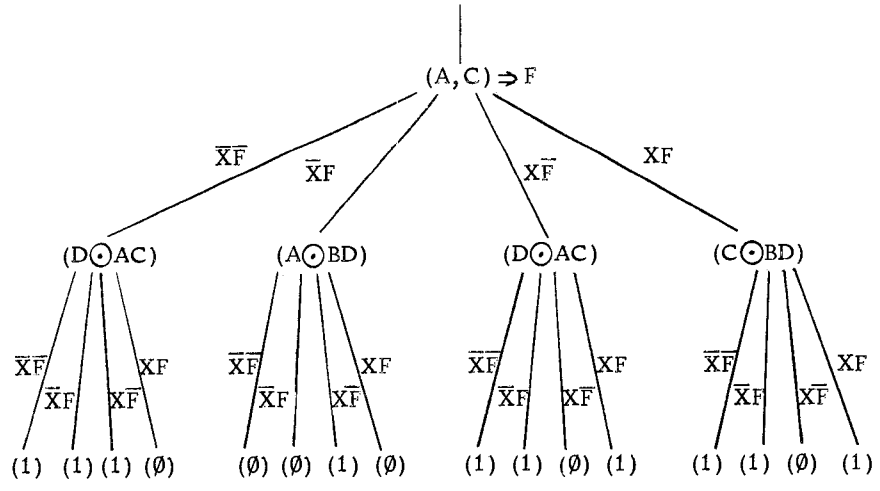
Figure 15. (a) Expanded flow table of M1.
(b) Predecessor tree of (a).

	$\bar{X}\bar{F}$	$\bar{X}F$	$X\bar{F}$	XF	Z	F
A	—	C	—	B	0	1
B	D	—	D	—	0	0
C	—	D	—	C	1	1
D	C	—	A	—	1	0

(a)



(b)



(c)

Figure 16. (a) Expanded flow table of M3.
(b) Z-predecessor tree of (a).
(c) F-predecessor tree of (a).

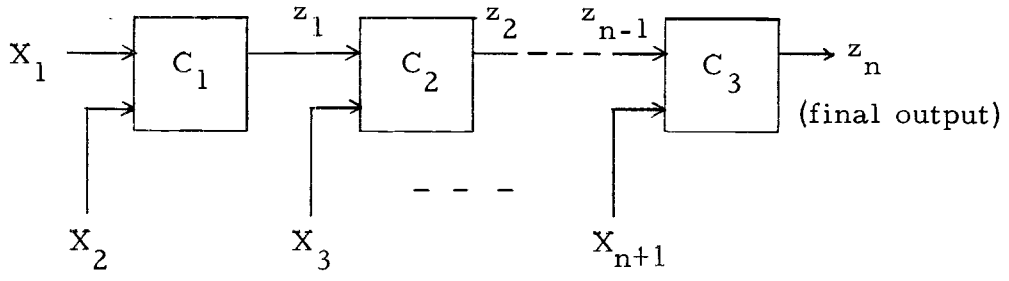
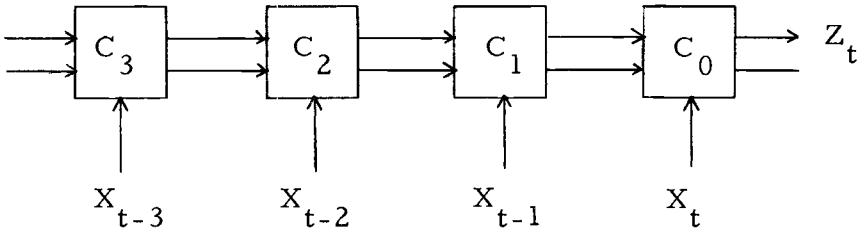


Figure 17. Maitra cascade. (z_n is the final output).

$$Z_t = \bar{X}_t\bar{X}_{t-1}X_{t-2} + \bar{X}_tX_{t-1} + X_t\bar{X}_{t-1}\bar{X}_{t-2}X_{t-3} + X_t\bar{X}_{t-1}X_{t-2}$$

(a)



(b)

Figure 18. (a) The difference equation for the output of M2.
(b) Two-rail cascade realization of (a).

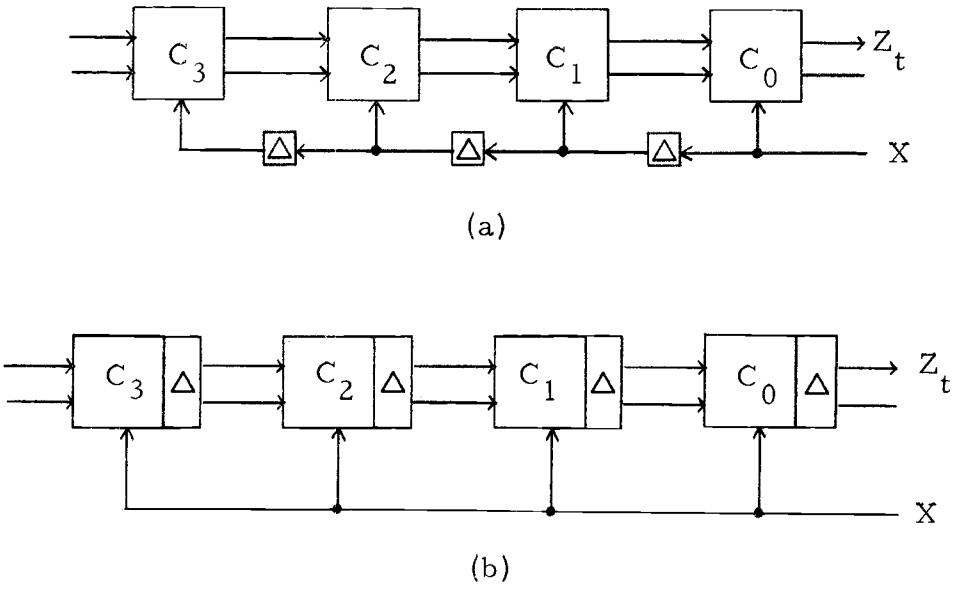


Figure 19. Two-rail cascade realization with (a) delayed input, and (b) delayed logic.

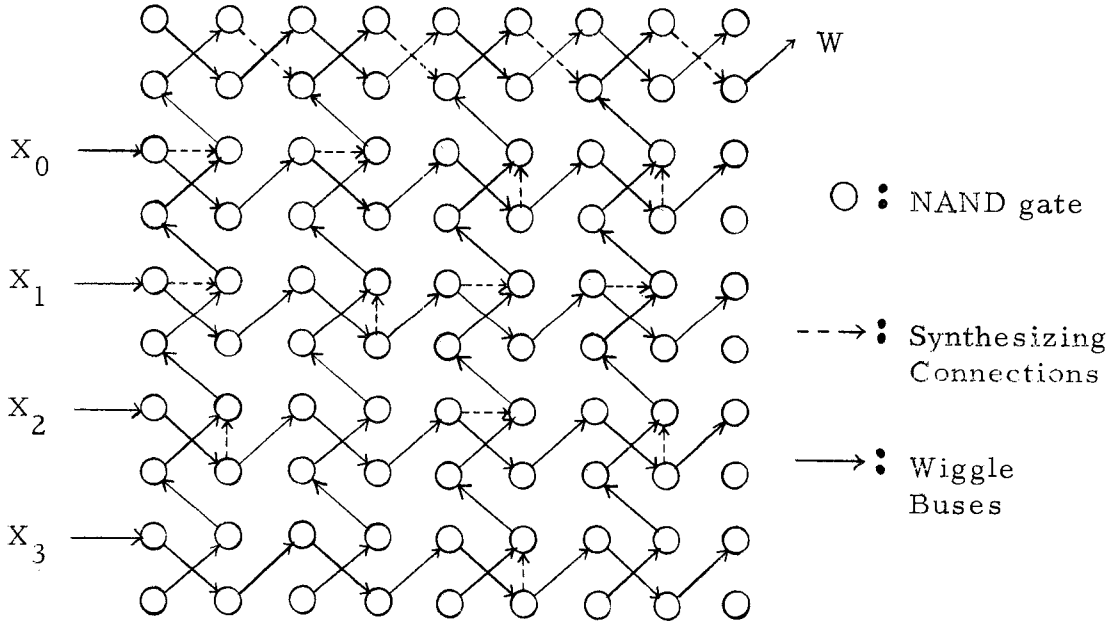
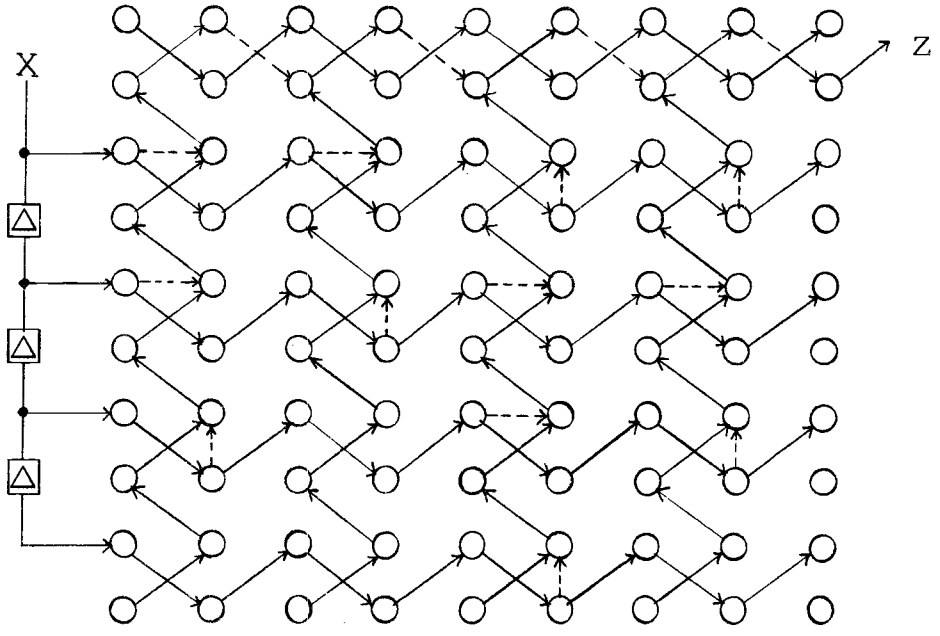
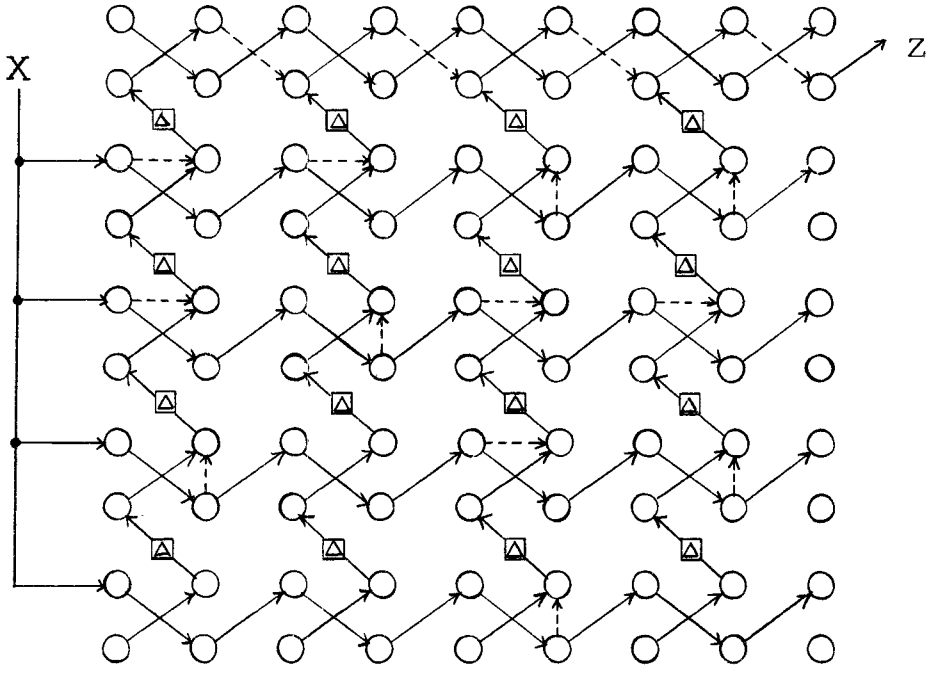


Figure 20. Cellular array synthesis of Equation 2.12 using Spandorfer's technique.



(a)



(b)

Figure 21. (a) Delayed input realization of M2.
(b) Delayed logic realization of M2.

$$Z_t = \bar{X}_t \bar{F}_{t-1} + \bar{X}_t F_{t-1} + X_t \bar{F}_{t-1} X_{t-1} F_{t-2} + Z_t F_{t-1} \bar{X}_{t-1} \bar{F}_{t-2} + X_t F_{t-1} \bar{X}_{t-1} F_{t-2} + X_t F_{t-1} X_{t-1} F_{t-2}$$

$$F_t = \bar{X}_t \bar{F}_{t-1} \bar{X}_{t-1} \bar{F}_{t-2} + \bar{X}_t \bar{F}_{t-1} \bar{X}_{t-1} F_{t-2} + \bar{X}_t \bar{F}_{t-1} X_{t-1} \bar{F}_{t-2} + \bar{X}_t F_{t-1} X_{t-1} \bar{F}_{t-2} + X_t \bar{F}_{t-1} \bar{X}_{t-1} \bar{F}_{t-2}$$

$$+ X_t \bar{F}_{t-1} \bar{X}_{t-1} F_{t-2} + X_t \bar{F}_{t-1} X_{t-1} \bar{F}_{t-2} + X_t F_{t-1} \bar{X}_{t-1} \bar{F}_{t-2} + X_t F_{t-1} \bar{X}_{t-1} F_{t-2} + X_t F_{t-1} X_{t-1} F_{t-2}.$$

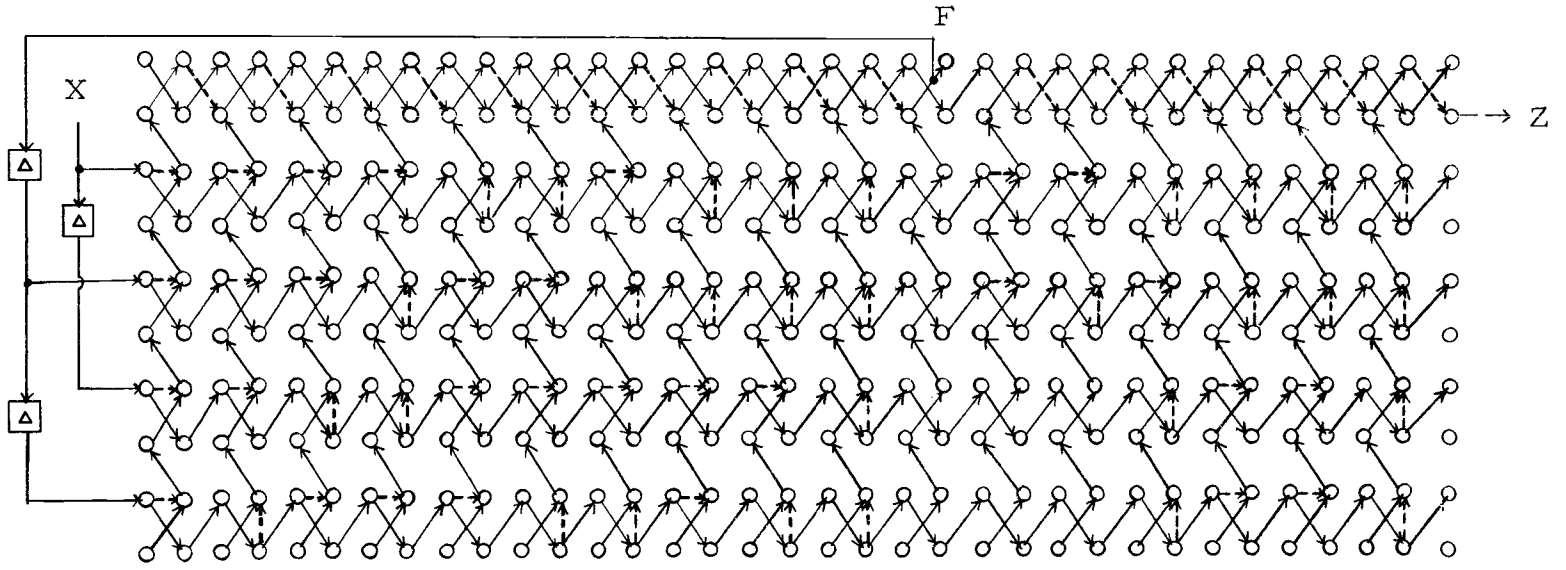
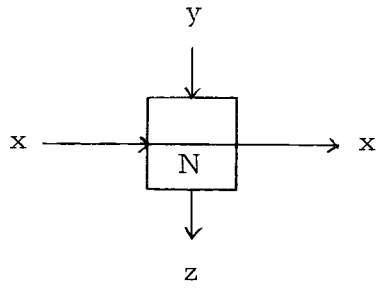


Figure 22 (a) Difference equation characterizing M3.
(b) Array realization of M3 using delayed input.



N	z
0	1
1	$\overline{y}$
2	$\overline{x} + \overline{y}$
3	$\overline{x}\overline{y}$
4	$x + y$
5	$x\overline{y}$
6	$x \oplus y$
7	0
F	x = S, y = R

Figure 23. Functional description of a cutpoint cell.

$$Z_t = \overline{X}_t \overline{X}_{t-1} X_{t-2} + \overline{X}_t X_{t-1} + X_t \overline{X}_{t-1} \overline{X}_{t-2} X_{t-3} + X_t \overline{X}_{t-1} X_{t-2}$$

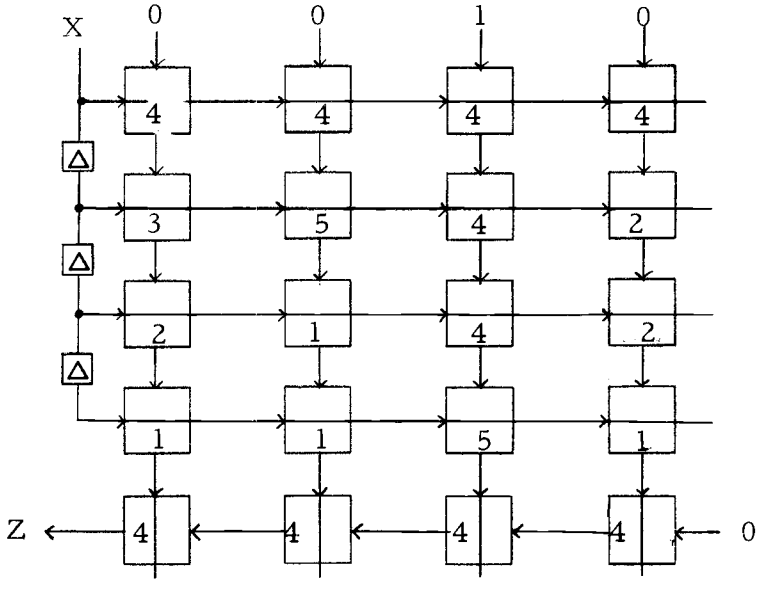


Figure 24. Cutpoint cellular array synthesis of

$$Z_t = \bar{X}_t \bar{Z}_{t-1} \bar{X}_{t-1} \bar{Z}_{t-2} + \bar{X}_t \bar{Z}_{t-1} \bar{X}_{t-1} Z_{t-2} + \bar{X}_t \bar{Z}_{t-1} X_{t-1} \bar{Z}_{t-2}$$
$$+ \bar{X}_t Z_{t-1} X_{t-1} \bar{Z}_{t-2} + X_t \bar{Z}_{t-1} \bar{X}_{t-1} \bar{Z}_{t-2} + X_t \bar{Z}_{t-1} \bar{X}_{t-1} Z_{t-2}$$
$$+ X_t \bar{Z}_{t-1} X_{t-1} \bar{Z}_{t-2} + X_t Z_{t-1} \bar{X}_{t-1} \bar{Z}_{t-2} + X_t Z_{t-1} \bar{X}_{t-1} Z_{t-2}$$
$$+ X_t Z_{t-1} X_{t-1} Z_{t-2}.$$

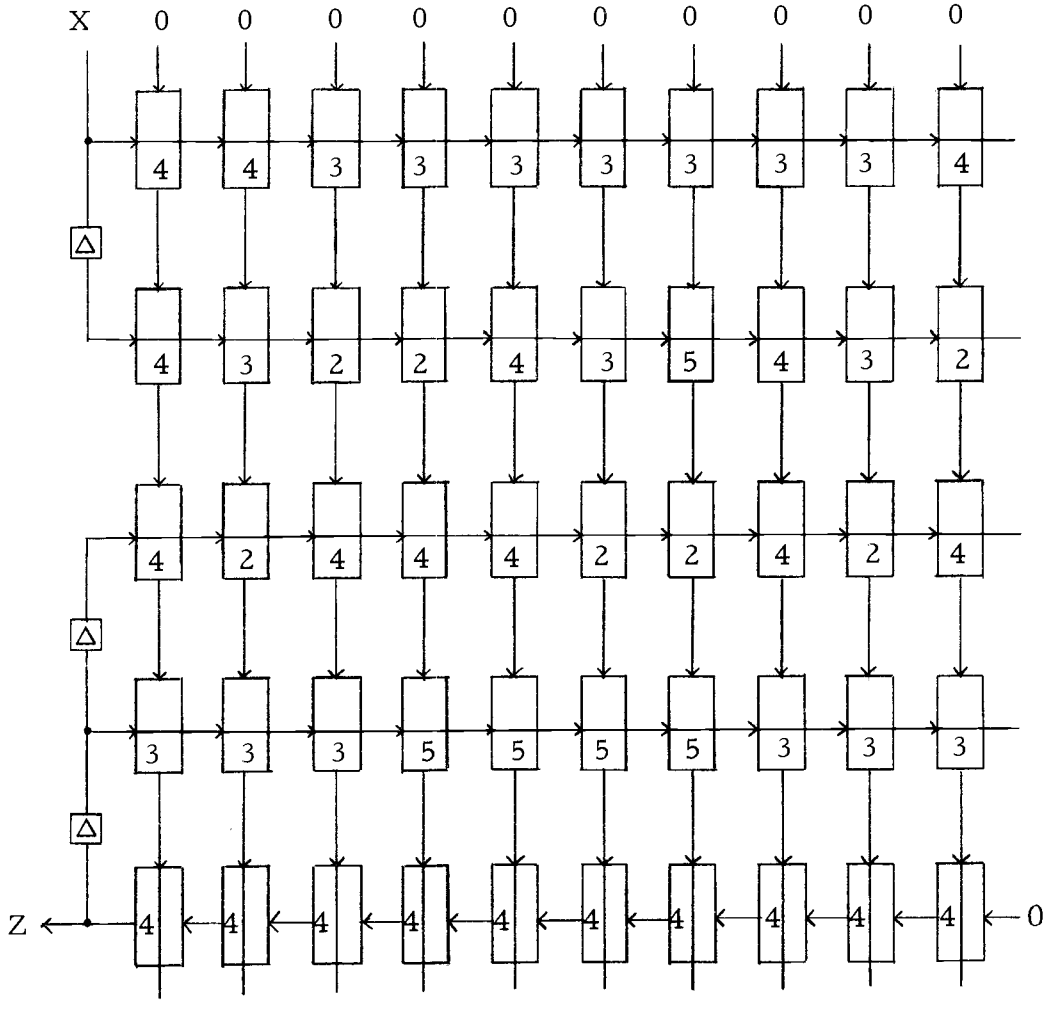


Figure 25. Cutpoint cellular array synthesis of M1.

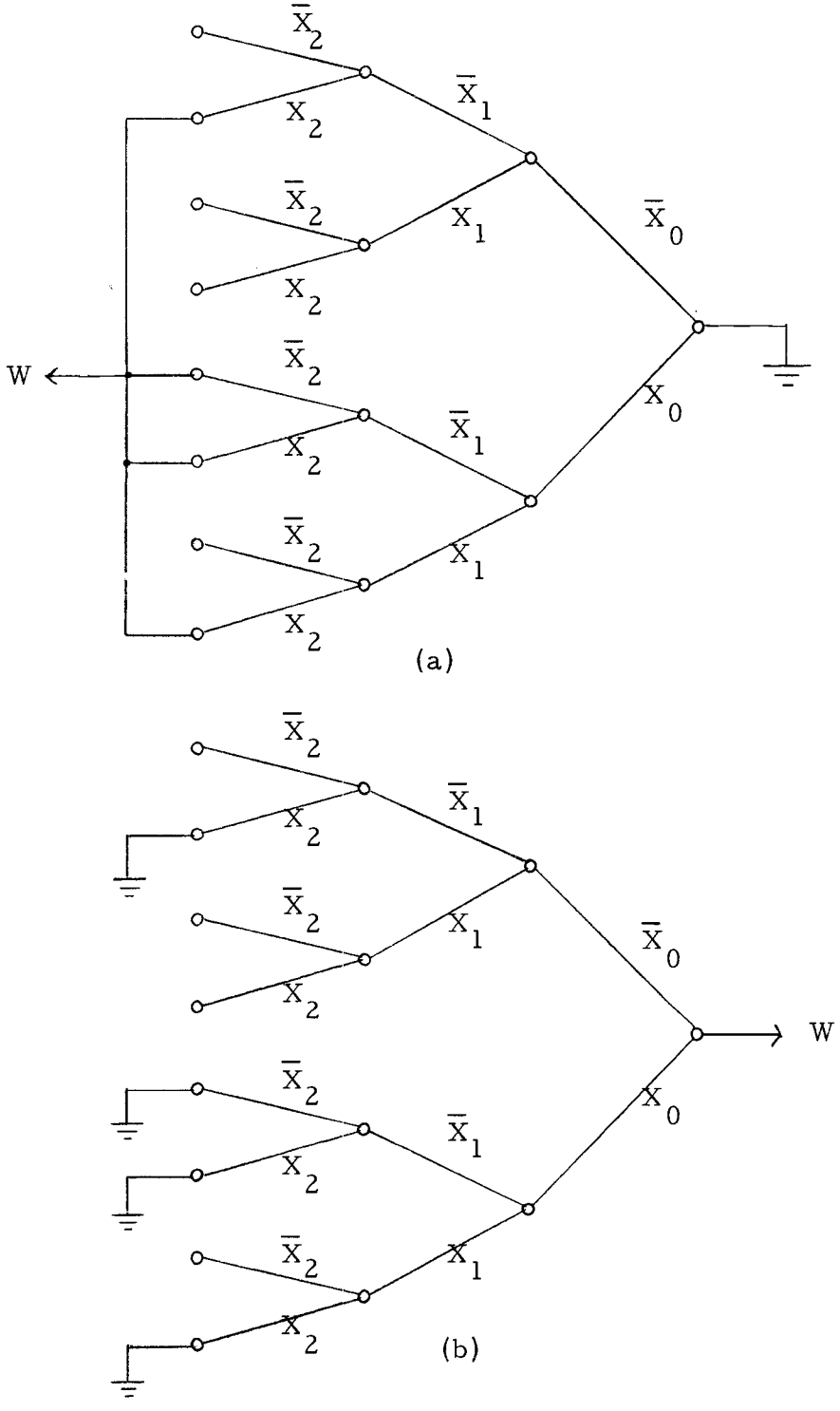


Figure 26. (a) Tree realization of Equation 2.14.
(b) Another way of tree realization of Equation 2.14.

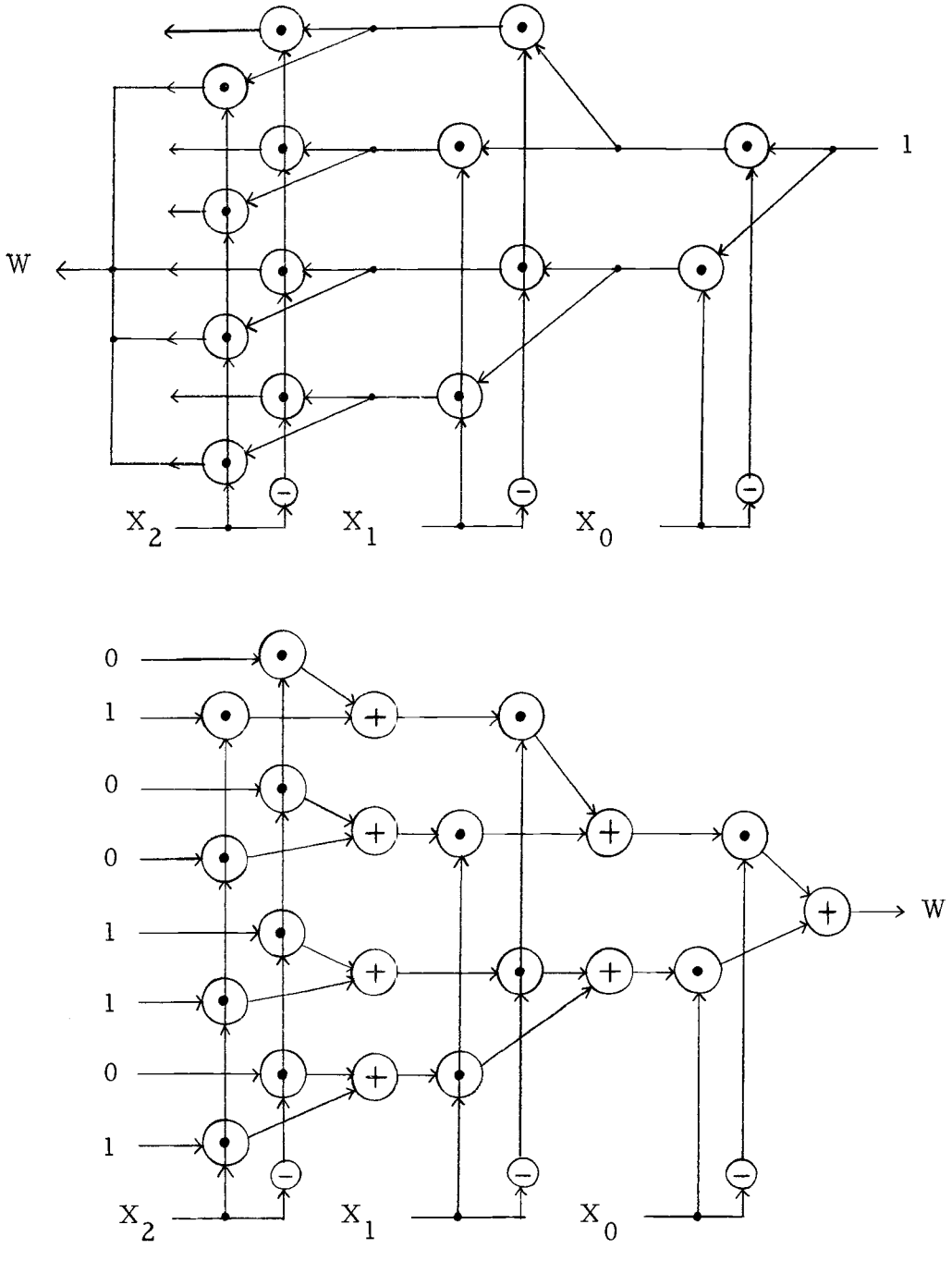


Figure 27. Two ways of realizing Equation 2.14 using gate circuits.

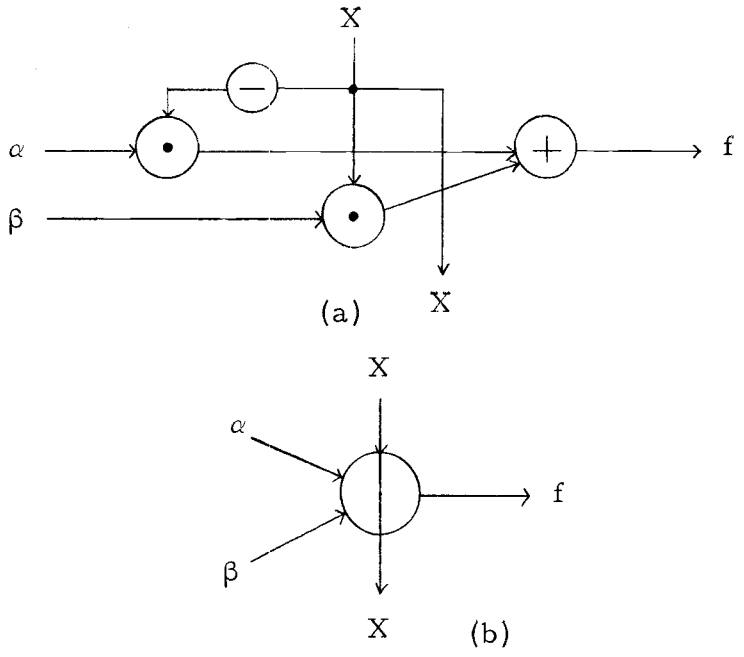


Figure 28. (a) Detailed gate circuit of a cell in a delayed input tree structure.
(b) The symbol represents the gate network of (a).

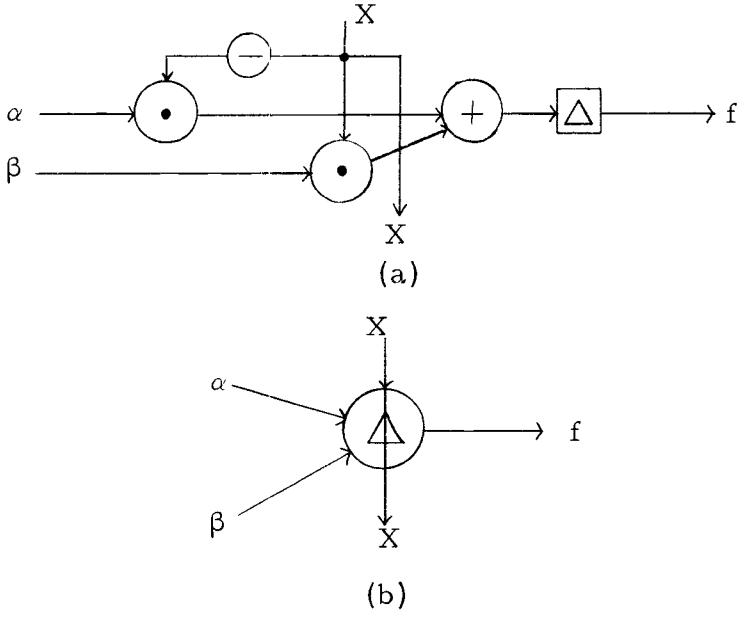
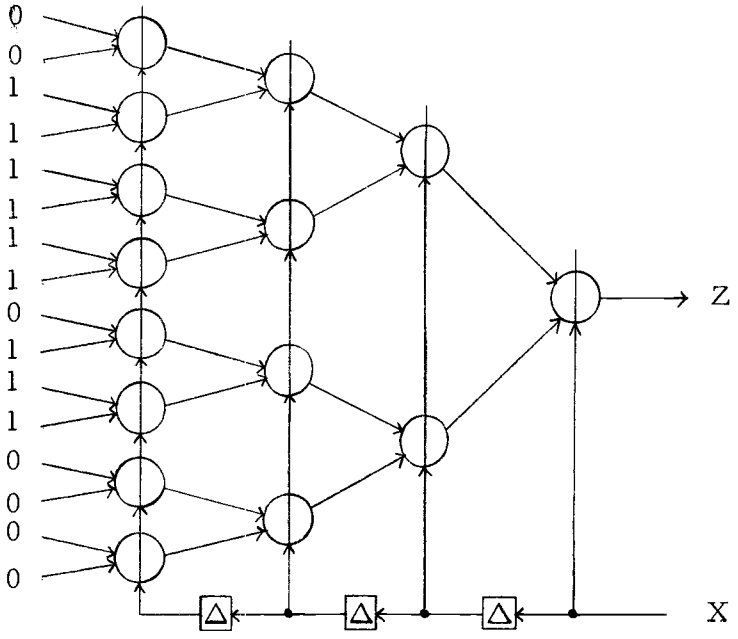


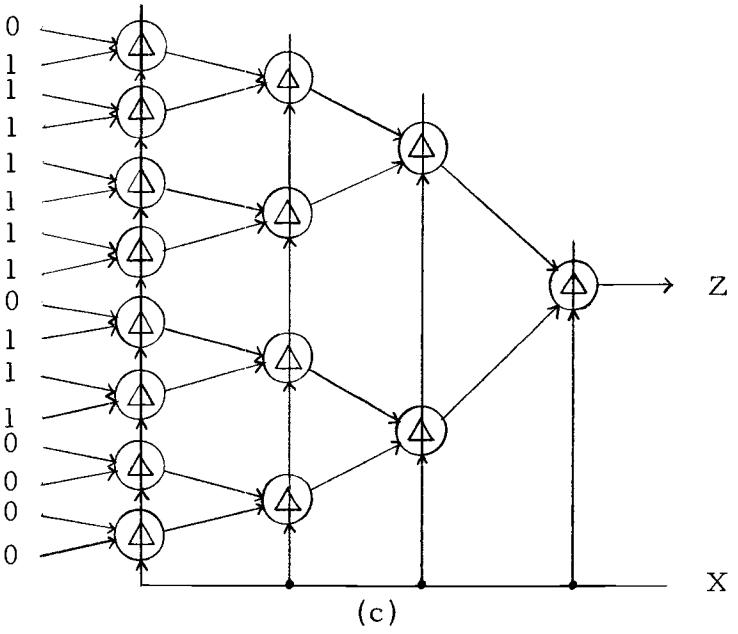
Figure 29. (a) Detailed gate circuit of a cell in a delayed logic tree structure.
(b) The symbol represents the gate network of (a).

$$Z_t = \bar{X}_t \bar{X}_{t-1} X_{t-2} + \bar{X}_t X_{t-1} + X_t \bar{X}_{t-1} \bar{X}_{t-2} X_{t-3} + X_t \bar{X}_{t-1} X_{t-2}$$

(a)

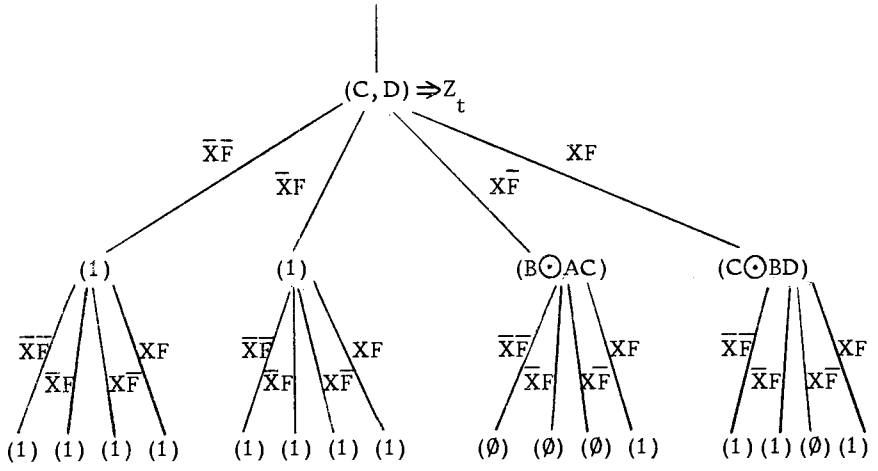


(b)

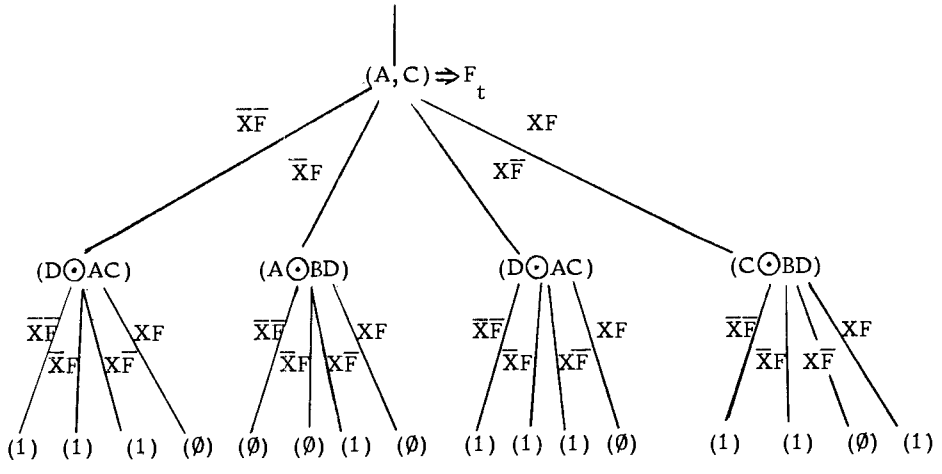


(c)

Figure 30. (a) Output equation of M2.
(b) Delayed input tree realization of M2.
(c) Delayed logic tree realization of M2.

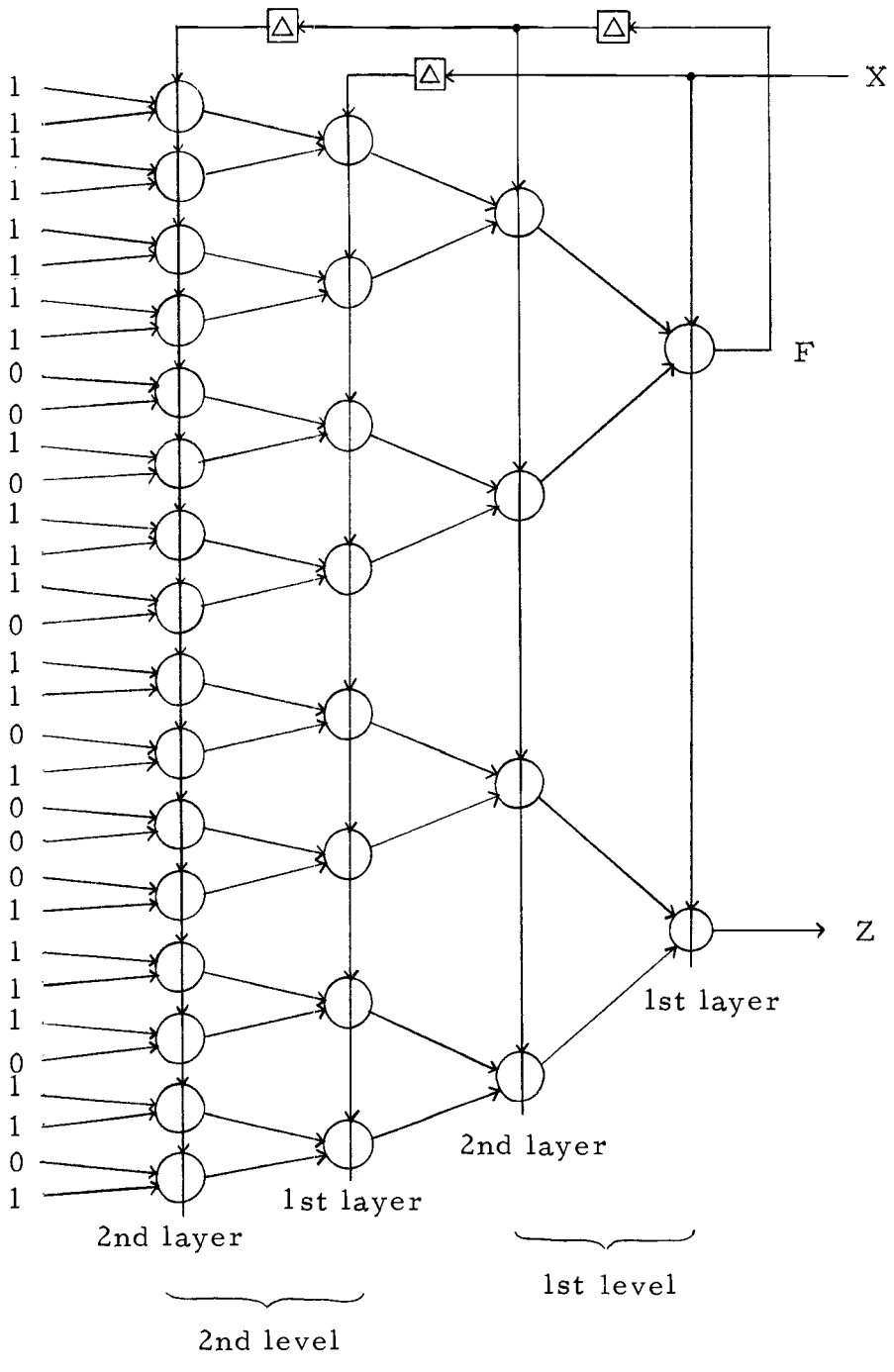


(a)



(b)

(Continued)



(c)

Figure 31. (a) Predecessor tree for Z of M3.
(b) Predecessor tree for F of M3.
(c) Tree realization of M3.

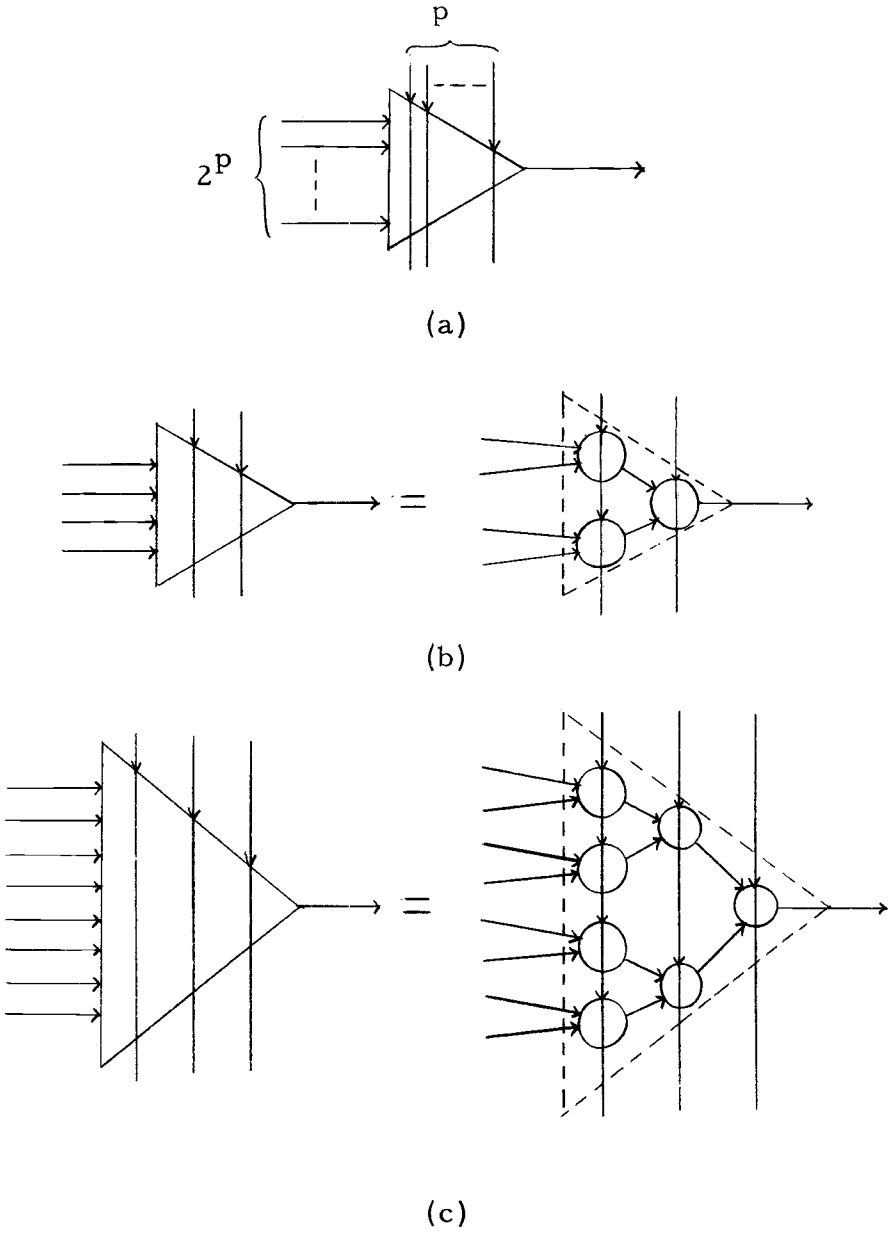


Figure 32. (a) Symbol for a super cell.
(b) Gate structure for a 2-input super cell.
(c) Gate structure for a 3-input super cell.

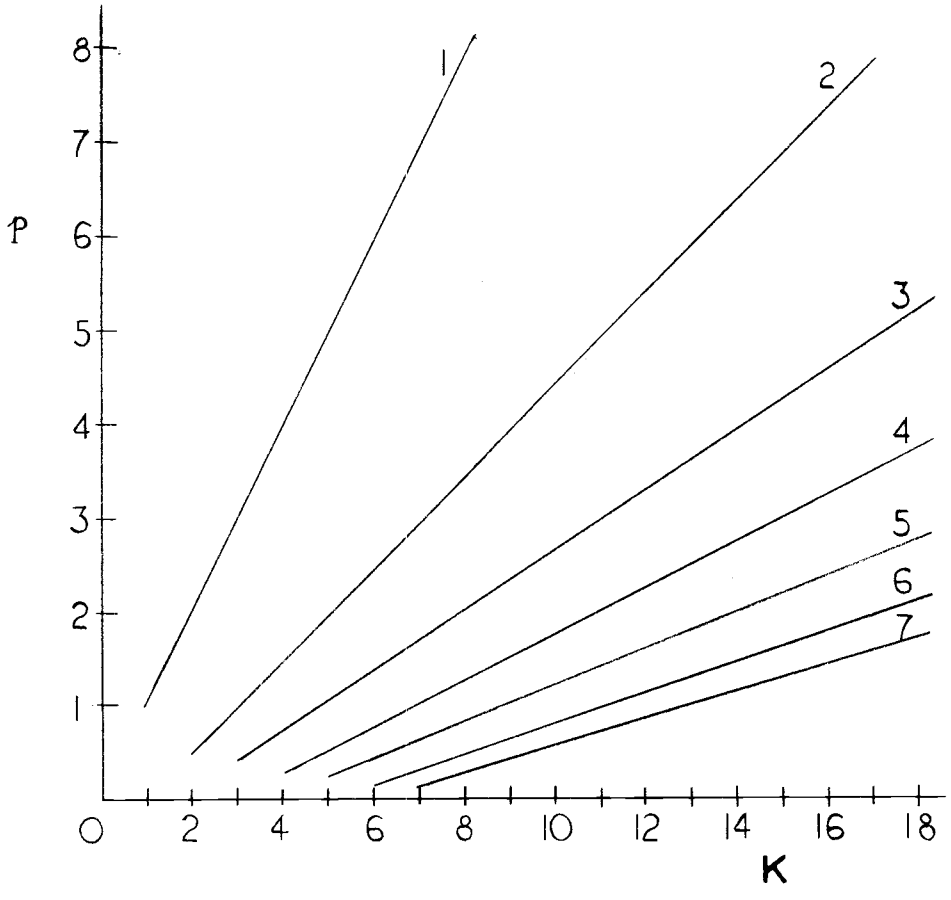


Figure 33. Tradeoff graph.
(K = original memory of the feedback free machine.
p = original number of input variables.
Numbers on slanted lines indicate the number of K being decreased by having a feedback. If the original p and K are such that their intersection lies above the line, then N would be smaller by having that feedback).

	i_1	i_2	Z
A	A	B	0
B	B	A	1

(a)

	A	B
A	i_1	i_2
B	i_2	i_1

(b)

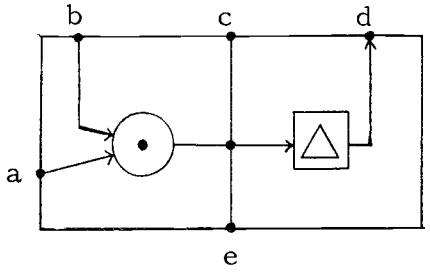
	A	B
A	i_1	0
B	0	i_1

(c)

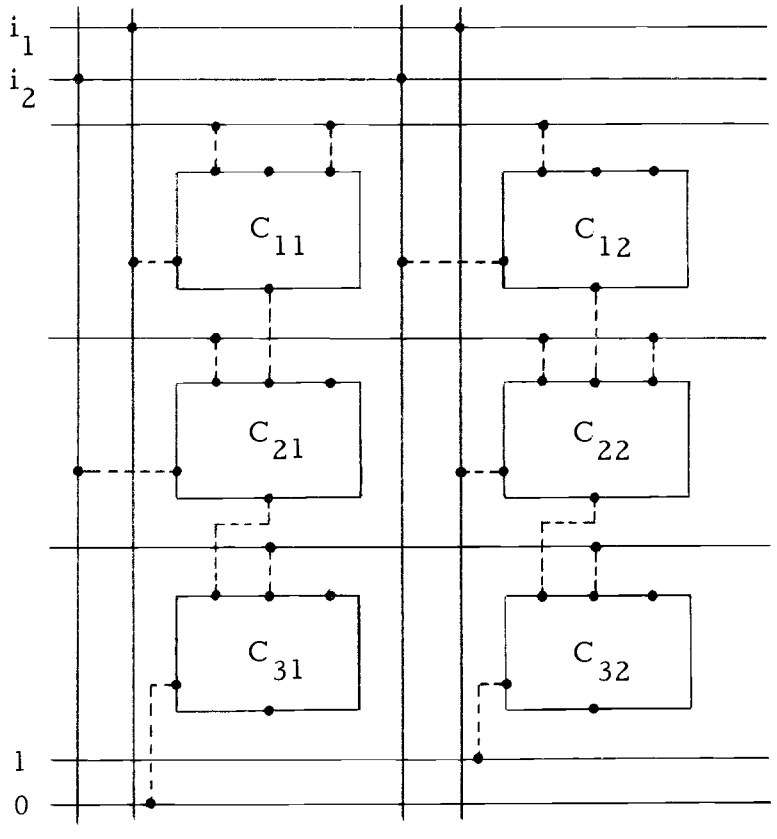
	A	B
A	0	i_2
B	i_2	0

(d)

Figure 34. (a) Flow table of M4.
(b) Transition matrix of M4.
(c) i_1-transition matrix of M4.
(d) i_2-transition matrix of M4.



(a)



(b)

Figure 35. (a) Cell structure.
(b) Matrix synthesis of M4.

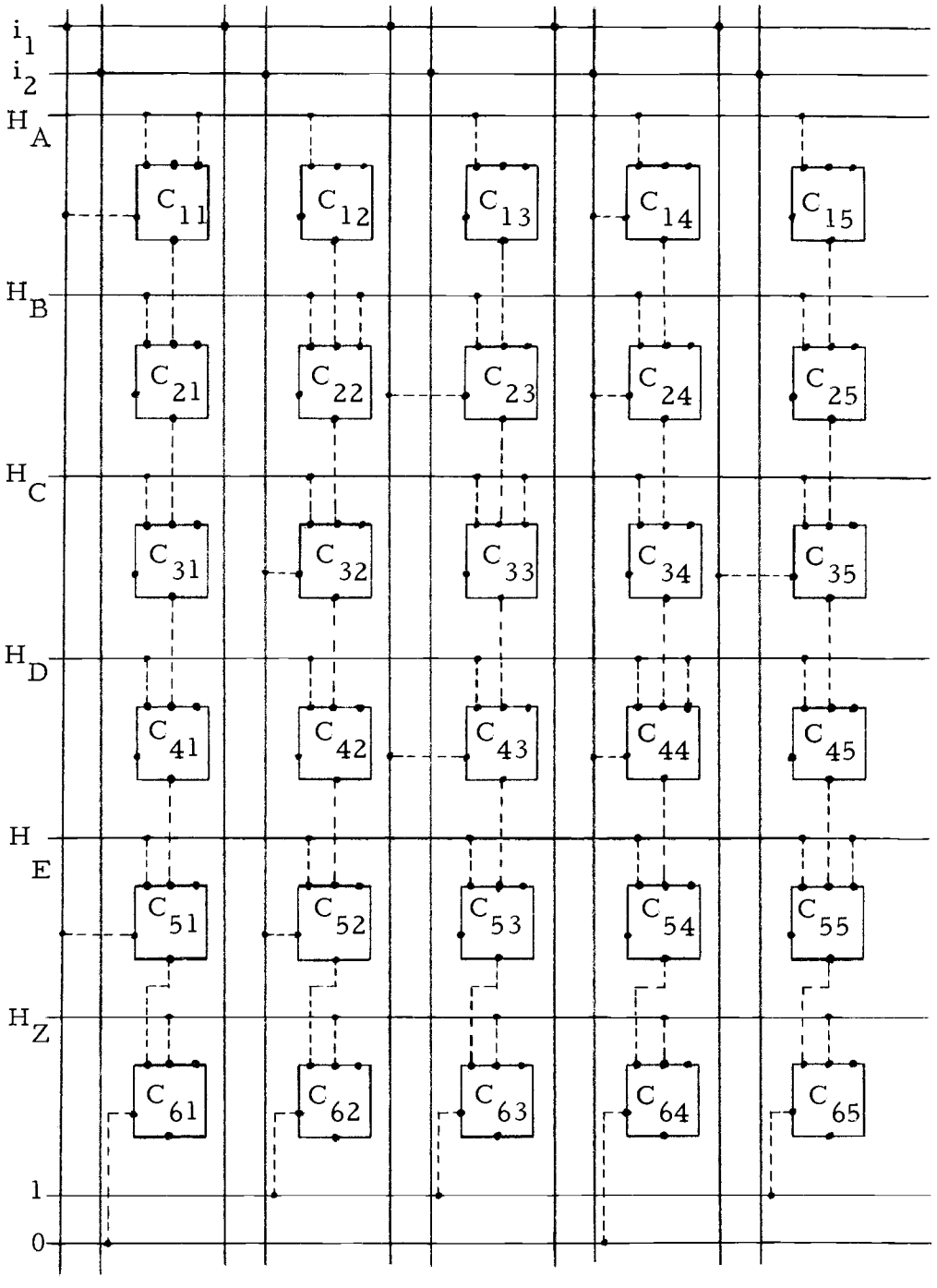


Figure 36. Matrix synthesis of M2.

	i_1	i_2	Z
A	A	B	1
B	B	C	0
C	C	A	0
D	D	E	0
E	D	E	1

Figure 37. A machine whose behavior depends on its starting state.

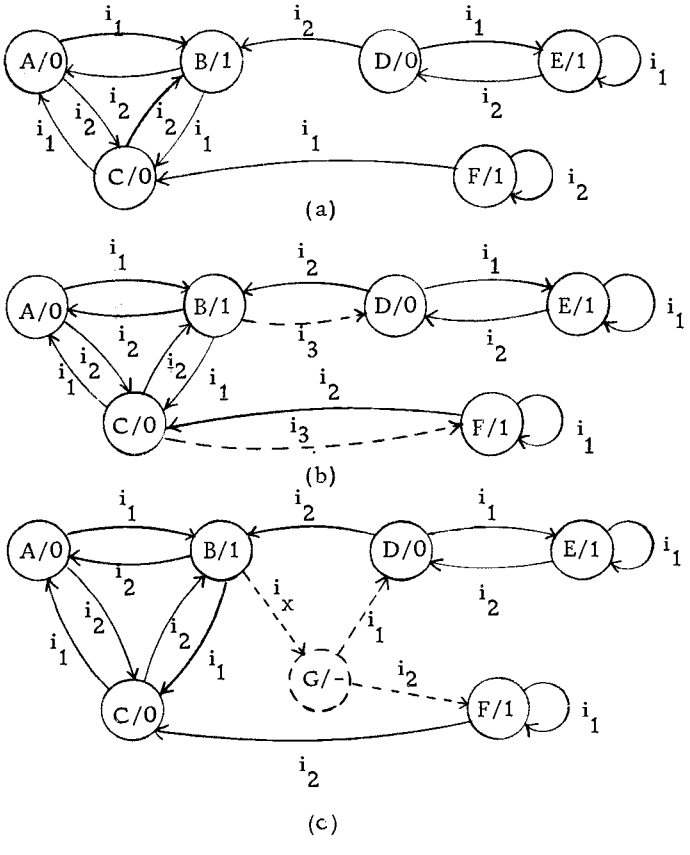


Figure 38. (a) State diagram of a machine with transient states.
(b) Added input i_3 makes transient states D, E and F recoverable.
(c) Added state G makes transient states D, E and F recoverable.

	i_1	i_2	Z
A	C	D	1
B	E	D	0
C	C	A	1
D	C	D	0
E	A	C	0

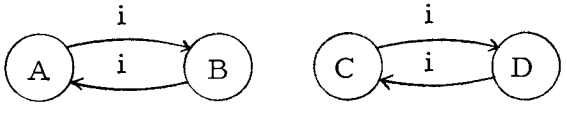
(a)

	i_1	i_2	i_3	i_4	Z
A	C	D	B	E	1
B	E	D	B	E	0
C	C	A	B	E	1
D	C	D	B	E	0
E	A	C	B	E	0

(b)

Figure 39. (a) Flow table of M5.
(b) Recoverable version of M5.

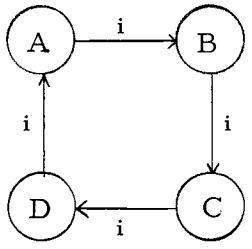
	i
A	B
B	A
C	D
D	C



2 loops

Figure 40. A machine with a permutation column but not a complete permutation column.

	i
A	B
B	C
C	D
D	A



1 loop

Figure 41. A machine with a complete permutation column.

	$\bar{Y}$	Y	Z
A	B	C	1
B	C	B	0
C	C	C	1
D	B	B	0
E	B	C	0
F	C	B	1

(a)

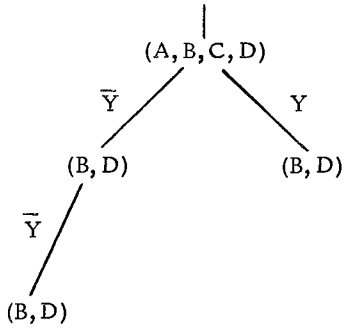
	$\bar{X}\bar{Y}$	$\bar{X}Y$	$X\bar{Y}$	XY	Z
A	B	C	—	—	1
B	C	B	A	D	0
C	C	C	E	F	1
D	B	B	—	—	0
E	B	C	—	—	0
F	C	B	—	—	1

(b)

Figure 42. (a) An example machine M.
(b) A recoverable version of (a).

	$\bar{Y}$	Y	Z
A	B	D	0
B	D	B	1
C	D	D	1
D	B	B	0

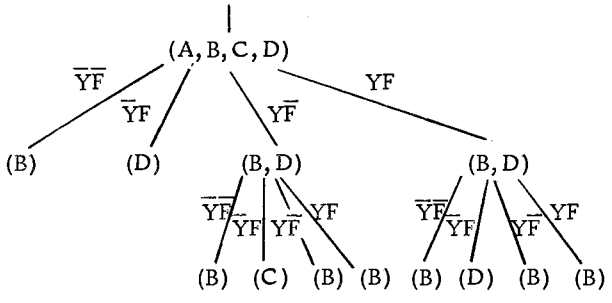
(a)



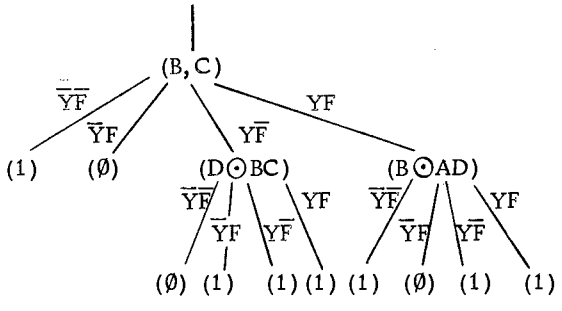
(b)

	$\bar{Y}\bar{F}$	$\bar{Y}F$	$Y\bar{F}$	YF	Z
A	B	—	D	—	0
B	—	D	—	B	1
C	—	D	—	D	1
D	B	—	B	—	0

(c)



(d)



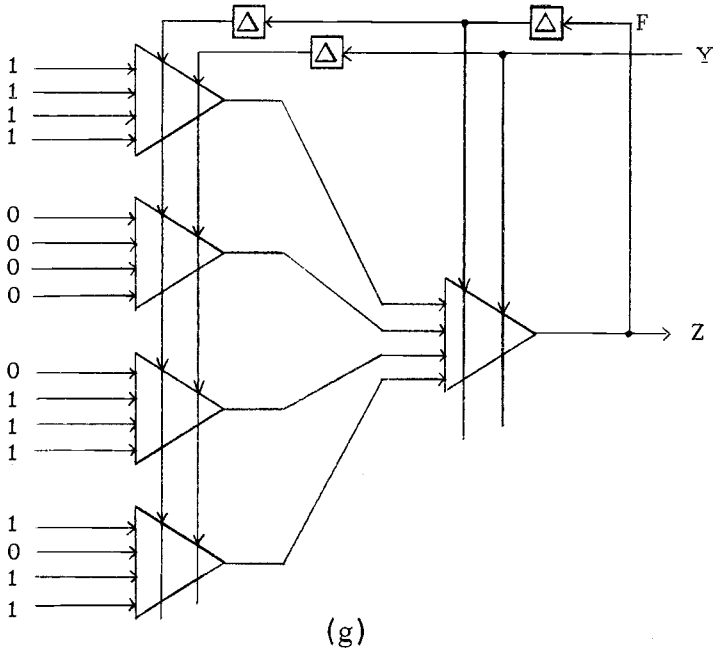
(e)

(Continued)

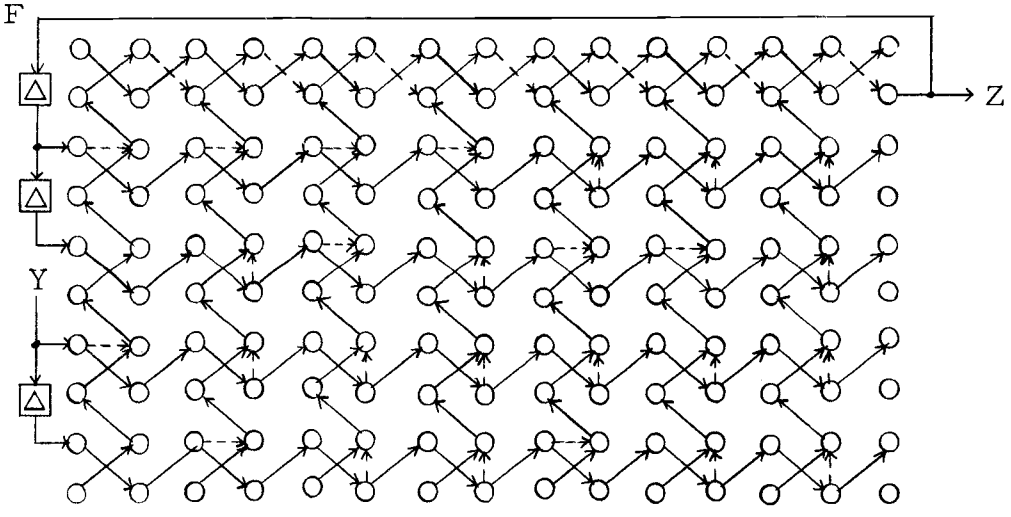
$$Z_t = \bar{Y}_t\bar{F}_{t-1} + Y_t\bar{F}_{t-1}\bar{Y}_{t-1}F_{t-2} + Y_t\bar{F}_{t-1}Y_{t-1}\bar{F}_{t-2} + Y_t\bar{F}_{t-1}Y_{t-1}F_{t-2}$$

$$+ Y_tF_{t-1}\bar{Y}_{t-1}\bar{F}_{t-2} + Y_tF_{t-1}Y_{t-1}\bar{F}_{t-2} + Y_tF_{t-1}Y_{t-1}F_{t-2}$$

(f)



(g)

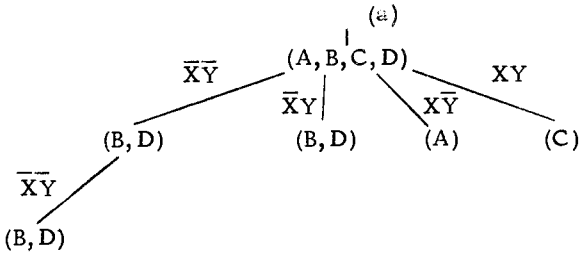


(h)

Figure 43. M6 and it synthesis procedures. (a) Flow table. (b) D-successor tree test. (c) M6 with the feedback input. (d) F-successor tree test. (e) The predecessor tree. (f) Output equation. (g) Tree realization. (h) Array realization.

	$\bar{X}\bar{Y}$	$\bar{X}Y$	$X\bar{Y}$	XY	Z
A	B	D	A	C	0
B	D	B	A	C	1
C	D	D	A	C	1
C	B	B	A	C	0

(a)

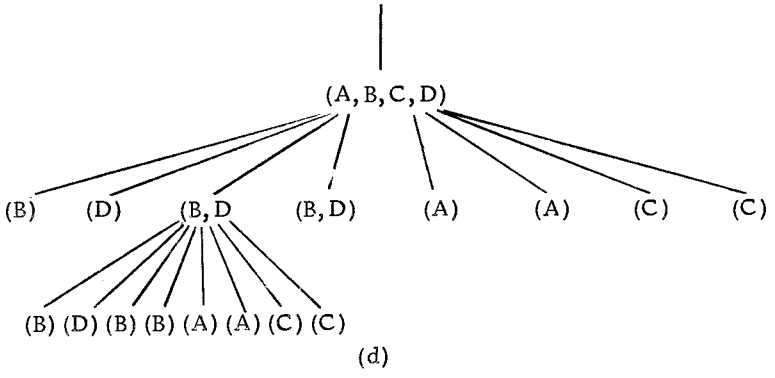


(b)

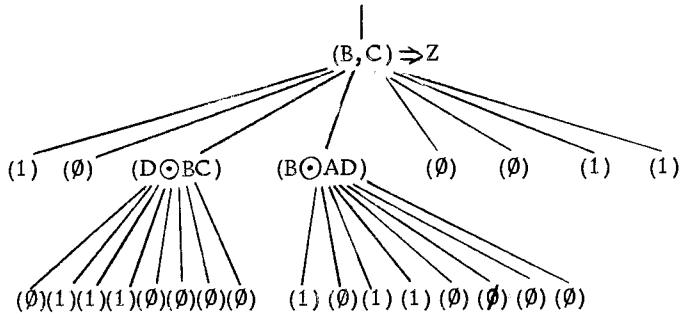
	$\bar{X}\bar{Y}\bar{F}$	$\bar{X}\bar{Y}F$	$\bar{X}Y\bar{F}$	$\bar{X}YF$	$X\bar{Y}\bar{F}$	$X\bar{Y}F$	$XY\bar{F}$	XFY	Z
A	B	—	D	—	A	—	C	—	0
B	—	D	—	B	—	A	—	C	1
C	—	D	—	D	—	A	—	C	1
D	B	—	B	—	A	—	C	—	0

(c)

(Continued)



(d)

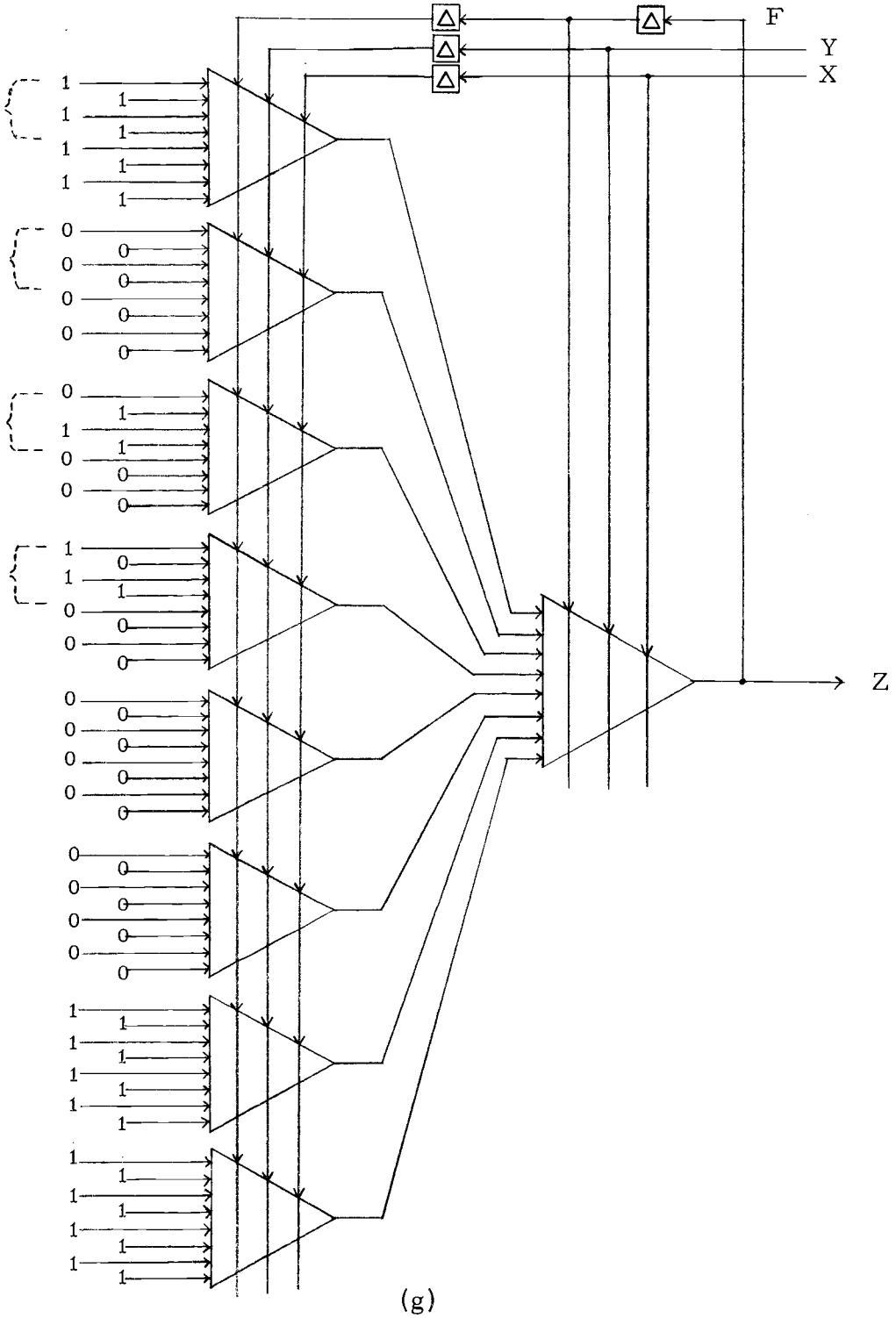


(e)

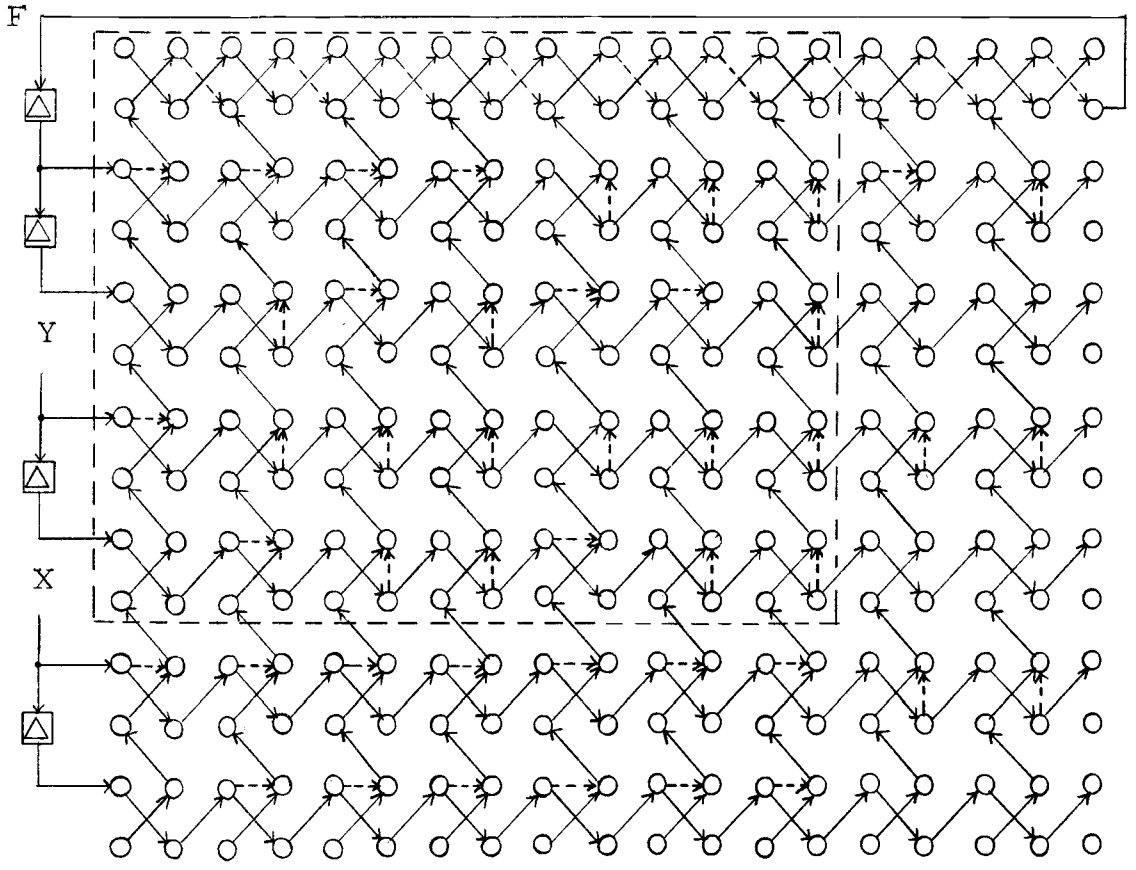
$$Z_t = \overline{X}_t \overline{Y}_t \overline{F}_{t-1} + \overline{X}_t Y_t \overline{F}_{t-1} (\overline{X}_{t-1} \overline{Y}_{t-1} F_{t-2} + \overline{X}_{t-1} Y_{t-1} \overline{F}_{t-2} + \overline{X}_{t-1} Y_{t-1} F_{t-2})$$
$$+ \overline{X}_t Y_t F_{t-1} (\overline{X}_{t-1} \overline{Y}_{t-1} \overline{F}_{t-2} + \overline{X}_{t-1} Y_{t-1} \overline{F}_{t-2} + \overline{X}_{t-1} Y_{t-1} F_{t-2})$$
$$+ X_t Y_t \overline{F}_{t-1} + X_t Y_t F_{t-1}$$

(f)

(Continued)



(g)



(h)

Figure 44. Recoverable version of M6 and its synthesis procedures.

(a) Flow table of M6'--recoverable version of M6.
(b) D-successor tree test.
(c) M6' with the feedback input.
(d) F-successor tree test.
(e) The predecessor tree.
(f) Output equation.
(g) Tree realization.
(h) Array realization.

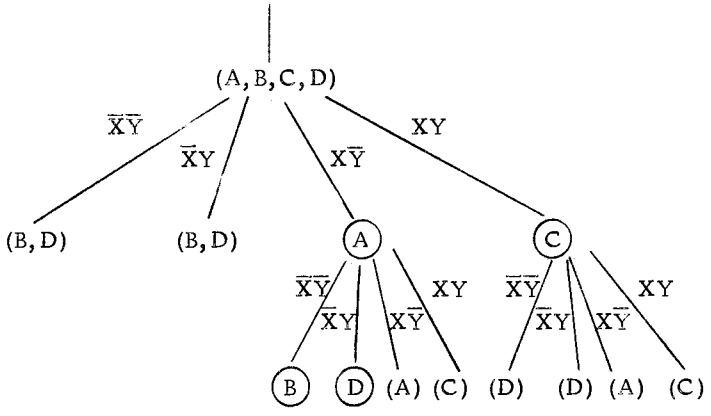


Figure 45. Recovering tree of $M6^{1}$.

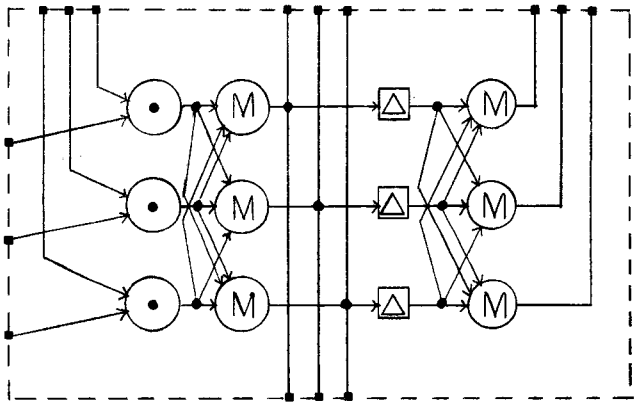


Figure 46. Cell structure for matrix arrays by employing triplicated redundancy technique and using majority gate as the decision making element.

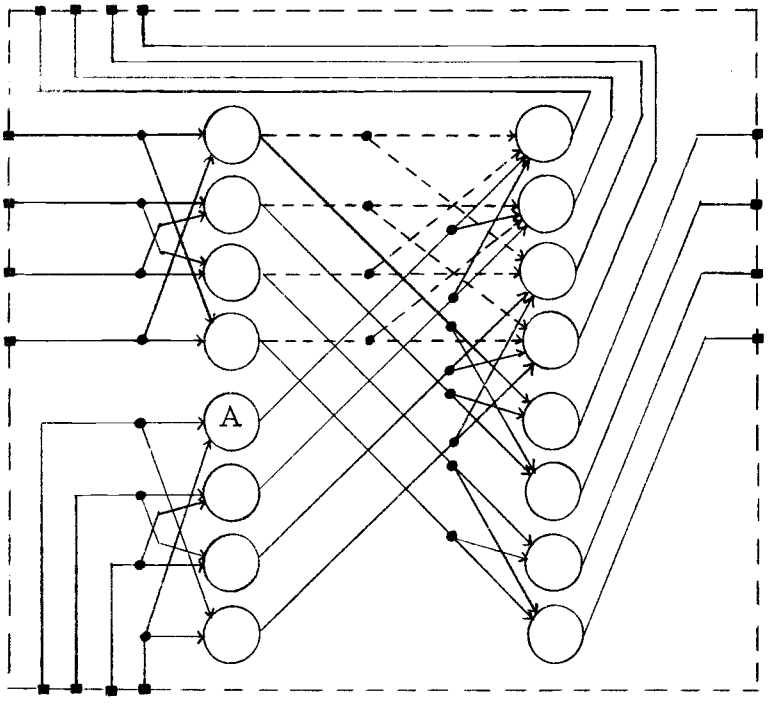
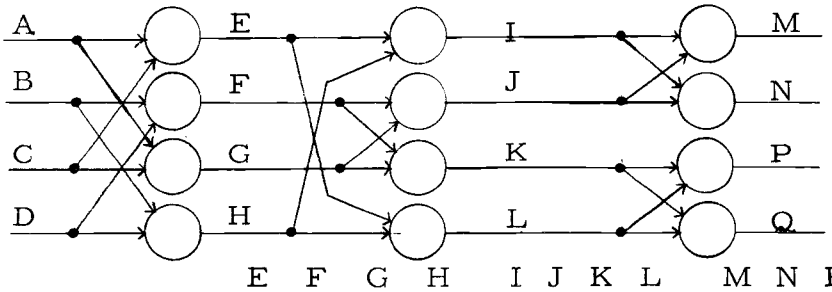


Figure 47. Quadded version of Spandorfer's array. (This network replaces four NAND-gate group. Dashed lines show possible synthesizing connections.)



	E	F	G	H	I	J	K	L	M	N	P	Q
Correct signals:	0	0	0	0	1	1	1	1	0	0	0	0
E is stuck at 1:	1	0	0	0	1	1	1	1	0	0	0	0
Correct signals:	1	1	1	1	0	0	0	0	1	1	1	1
E is stuck at 0:	0	1	1	1	1	0	0	0	1	1	1	1

All gates are NAND gates.

Figure 48. Error correcting mechanism in quadded logic.

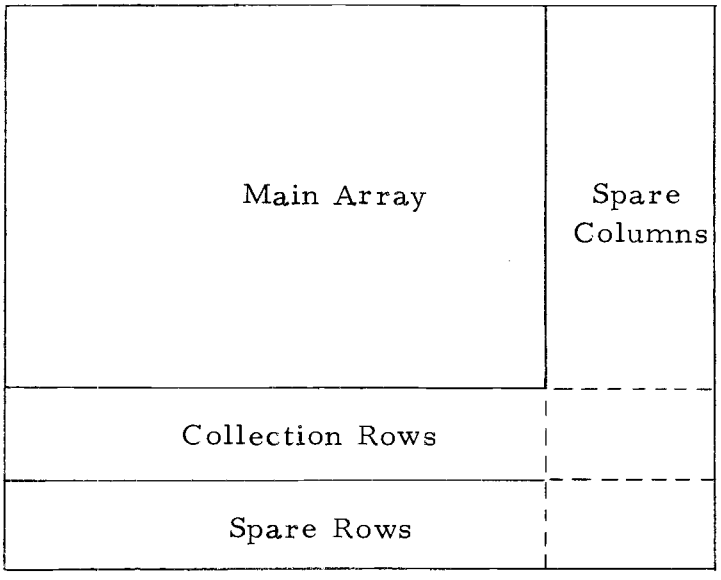


Figure 49. Cellular array with spare arrangement.

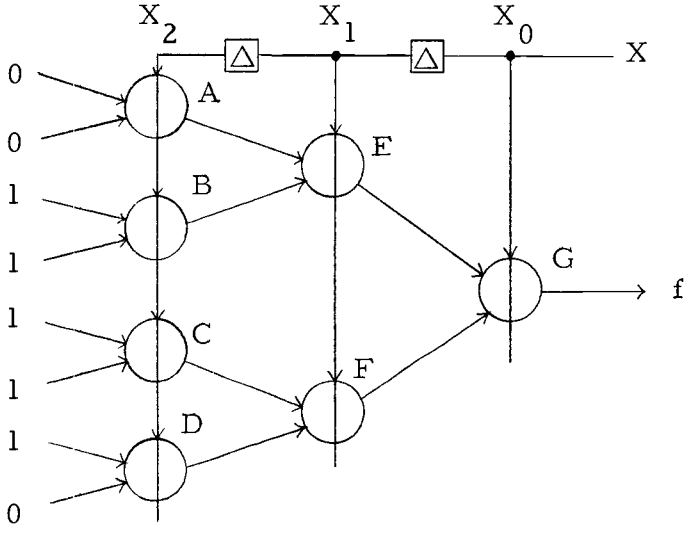


Figure 50. Tree structure for realizing the function $f = \bar{X}_0 X_1 + X_0 \bar{X}_1 X_2 + X_0 X_1 \bar{X}_2$.

X_2	X_1	X_0	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(a)

X_2	X_1	X_0	f	f_A	f_B	f_{C0}	f_{C1}	f_{D0}	f_{D1}	f_{E0}	f_{E1}	f_{F0}	f_{F1}	f_{G0}	f_{G1}
0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	1
0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	1
0	1	0	1	1	0	1	1	1	1	0	1	1	1	0	1
0	1	1	1	1	0	1	1	1	1	0	1	1	1	0	1
1	0	0	0	0	0	0	1	0	0	0	0	1	1	0	1
1	0	1	1	1	1	0	1	1	1	1	1	1	1	0	1
1	1	0	1	1	1	1	1	0	1	1	1	1	1	0	1
1	1	1	0	0	0	0	0	0	1	0	0	1	1	0	1

(b)

X_2	X_1	X_0	f	f_A	f_B	f_{C0}	f_{C1}	f_{D0}	f_{D1}	f_{E0}	f_{E1}	f_{F0}	f_{F1}	f_{G0}	f_{G1}
0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	1
0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	1
0	1	1	1	1	0	1	1	1	1	0	1	1	1	0	1
1	1	1	0	0	0	0	0	0	1	0	0	0	1	0	1
1	1	0	1	1	1	1	1	0	1	1	1	0	1	0	1
1	0	1	1	1	1	0	1	1	1	1	1	0	1	0	1
0	1	0	1	1	0	1	1	1	1	0	1	1	1	0	1
1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1

(c)

Figure 51. (a) Truth table describing the tree circuit of Figure 50.
(b) Possible functions for a single fault in Figure 50.
(c) Same as (b) with some row permutations.

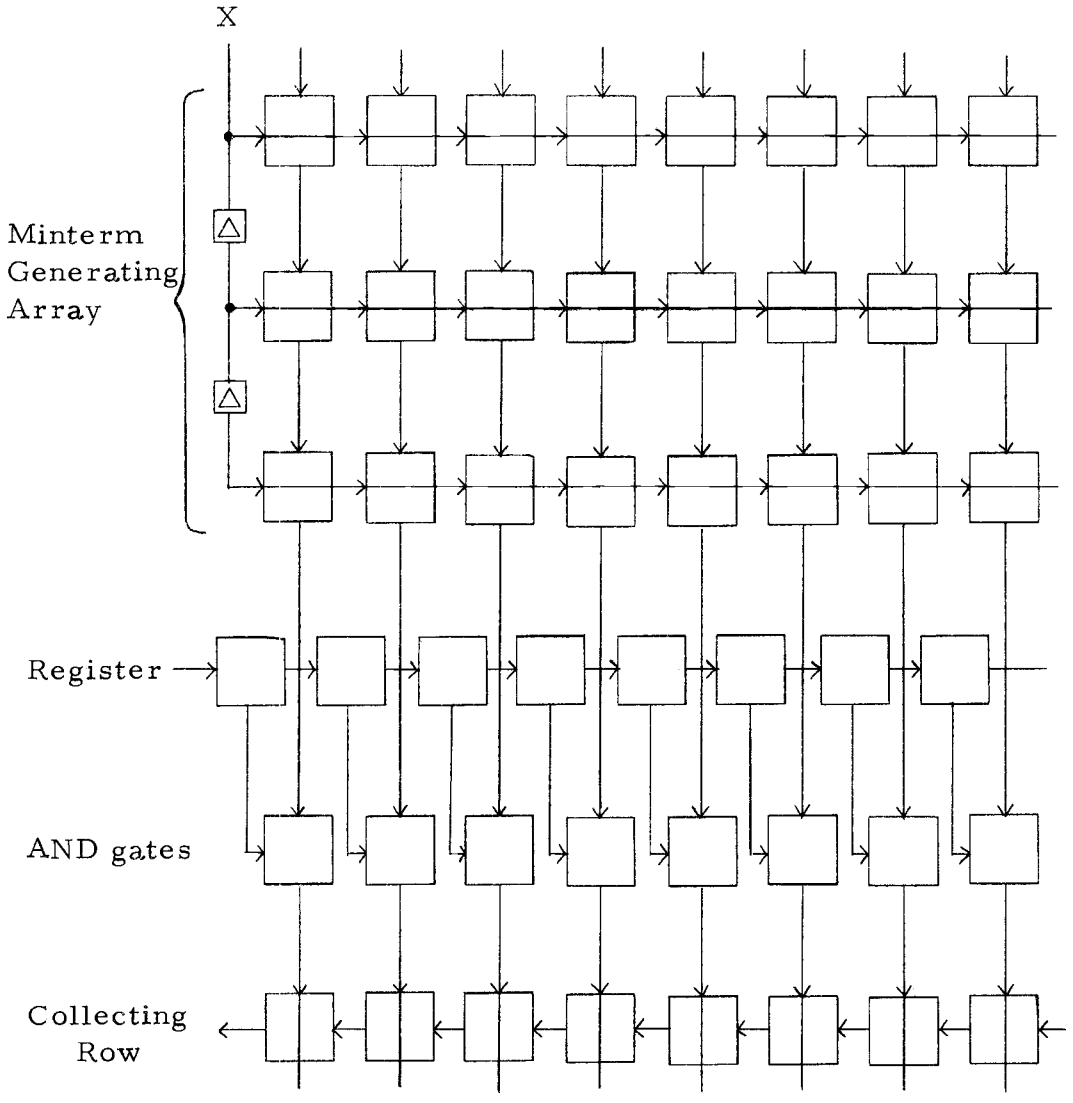
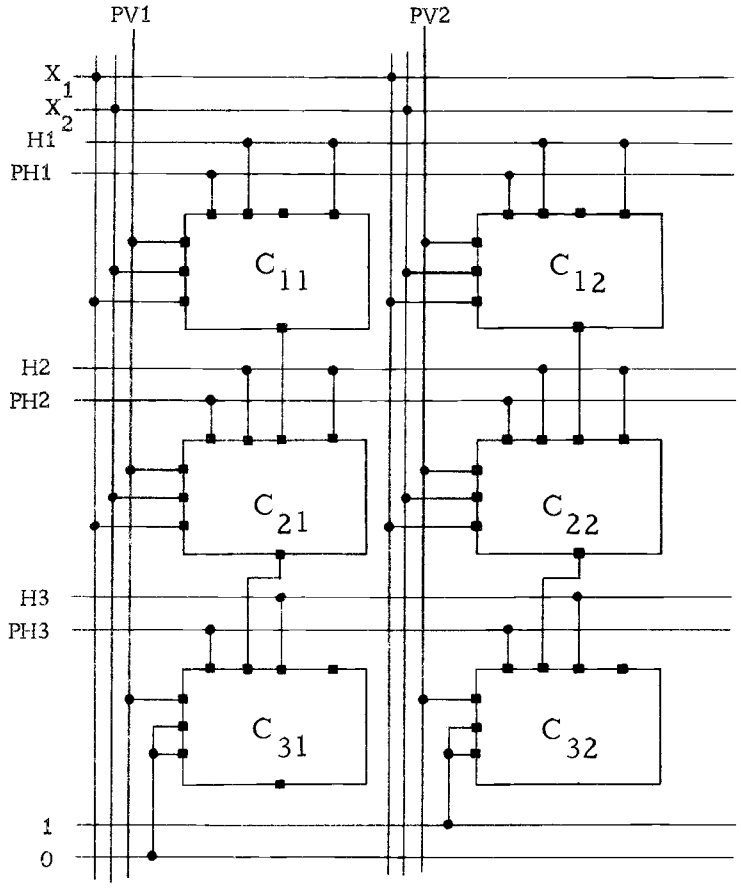
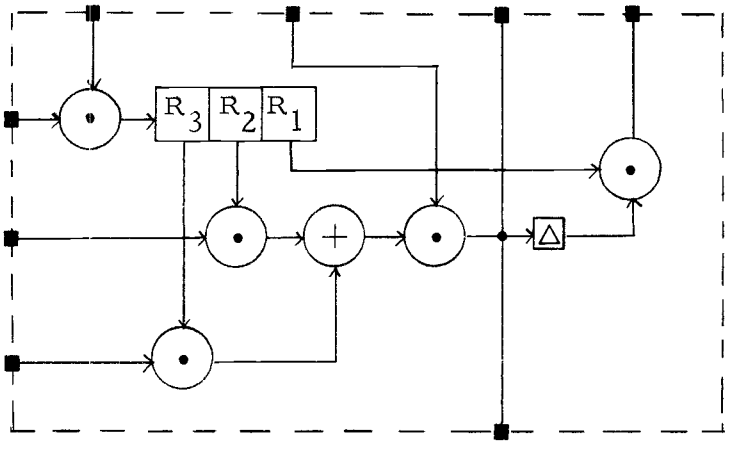


Figure 52. Programmable cut-point cellular array.



(a)



(b)

Figure 53. (a) Programmable matrix structure.
(b) The structure of cells used in (a).

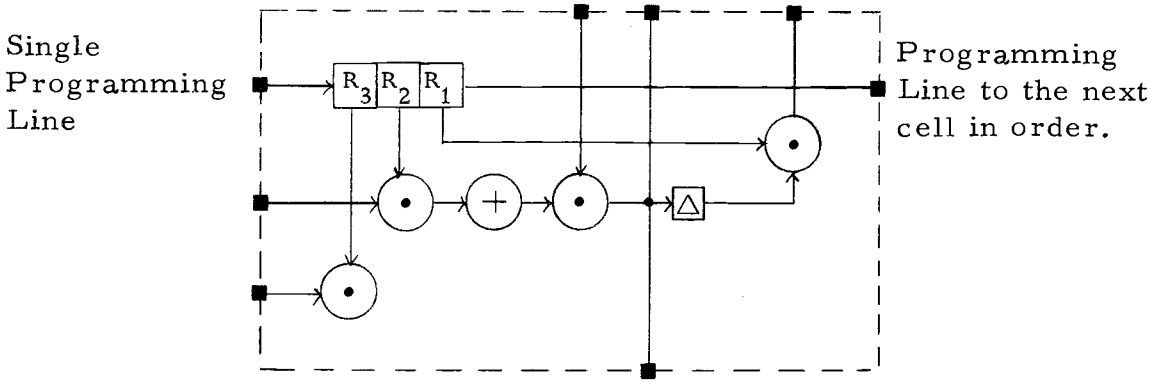
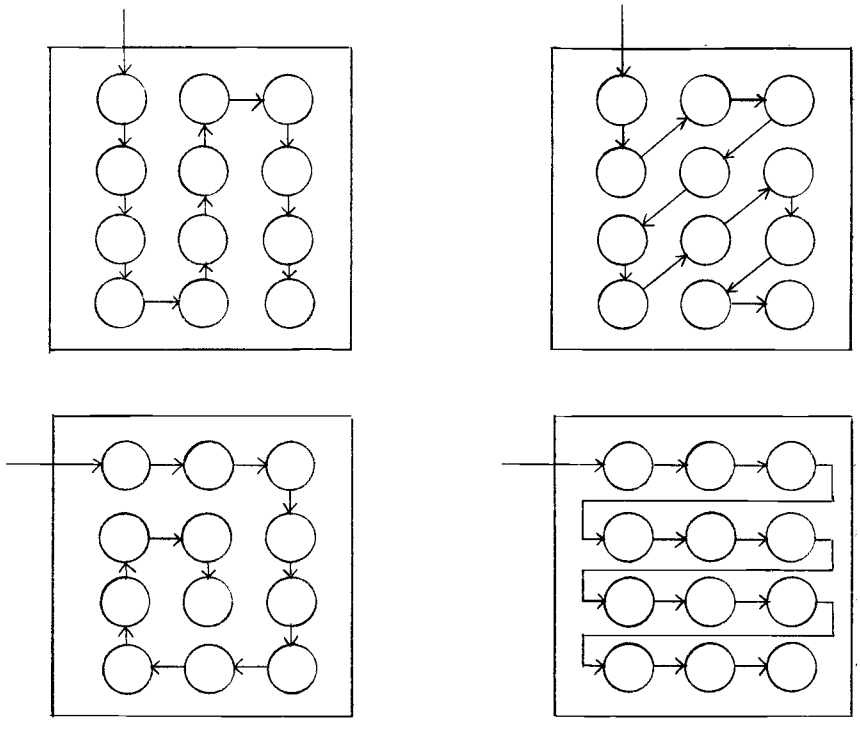


Figure 54. Cell structure used in linear select scheme of programming.



Each circle represents a cell

Figure 55. Some possible arrangements of the programming bus in linear select scheme.

BIBLIOGRAPHY

1. Brzozowski, J. A. On single-loop realizations of automata. In: IEEE Conference Recordings on Switching Circuit Theory and Logical Design, Ann Arbor, Michigan, 1965. Ann Arbor, 1965. p. 81-93.

2. Dorrough, D. C. A methodical approach to analyzing and synthesizing a self-repairing computer. IEEE Transactions on Computers C-18: 22-42. 1969.

3. Dvorak, V. A two-rail cascade synthesis of Boolean functions. IEEE Transactions on Computers C-17: 592-596. 1968.

4. Elspas, B. The theory of autononous linear sequential networks. IRE Transactions on Circuit Theory CT-6: 45-60. 1959.

5. Elspas, B., W. H. Kautz and H. S. Stone. Properties of cellular arrays for logic and storage. Menlo Park, Califonia, 1967. 55 p. (Stanford Research Institute. Report AFCRL-68-0005)

6. Elspas, B. and J. B. Turner. Theory of cellular logic networks and machines. Menlo Park, California, 1968. 97 p. (Stanford Research Institute. Report AFCRL-68-0148)

7. Friedman, Arthur D. Feedback in synchronous sequential switching circuits. IEEE Transactions on Electronic Computers EC-15: 354-367. 1966.

8. Gill, Arthur. Cascaded finite-state machines. IRE Transactions on Electronic Computers EC-10: 366-370. 1961.

9. Gill, Arthur. Introduction to the theory of finite-state machines. New York, McGraw-Hill, 1962. 207 p.

10. Hartmanis, J. and R. E. Sterns. Algebraic structure theory of sequential machines. Englewood Cliffs, New Jersey, Prentice-Hall, 1966. 211 p.

11. Hènnie, F. C. Finite-state models for logical machines. New York, Wiley, 1968. 466 p.

12. Kautz, W. H. Fault testing and diagnosis in a combinational digital circuits. IEEE Transactions on Computers C-17: 352-366. 1968.

13. Kohavi, A. and E. J. Smith. Decomposition of sequential machines. In: IEEE Conference Recordings on Switching Circuit Theory and Logical Design, Ann Arbor, Michigan, 1965. Ann Arbor, 1965. p. 52-61.

14. Krohn, K. B. and J. L. Rhodes. Algebraic theory of machines. I, prime decomposition theorem for finite semigroup and machines. American Mathematical Society Transactions 116: 450-464. 1965.

15. Low, P. R. and G. A. Maley. Flow table logic. Proceedings of the IRE 49: 221-228. 1961.

16. Maitra, K. K. Cascaded switching networks of two-input flexible cells. IRE Transactions on Electronic Computers EC-11: 136-143. 1962.

17. Minnick, R. C. and R. A. Short. Cellular linear-input logic. Menlo Park, California, 1964. (Stanford Research Institute. Report AFCRL-64-6)

18. Minnick, R. C. Cutpoint cellular logic. IEEE Transactions on Electronic Computers EC-13: 685-698. 1964.

19. Minnick, R. C. Survey of microcellular research. Menlo Park, California, 1966. 63 p. (Stanford Research Institute. Report AFCRL-66-475)

20 Perles, M., M. O. Rabin and E. Shamir. The Theory of definite automata. IEEE Transactions on Electronic Computers EC-12:233-243. 1963.

21. Short, R. A. The attainment of reliable digital systems through the use of redundancy--a survey. IEEE Computer Group News 2: 2-17. 1968.

22. Short, R. A. Two-rail cellular cascades. In: Proceedings of the Fall Joint Computer Conference of the American Federation of Information Processing Societies, Las Vegas, 1965. Vol. 27, part I. New York, 1965. p. 355-369.

23. Simon, J. M. A note on memory aspects of sequence transducers. IRE Transactions on Circuit Theory CT-6: 26-29. 1959.

24. Spandorfer, L. M. and A. B. Tonik. Planar interconnect logic. Proceedings of a Symposium on Microelectronics and Large Systems, Spartan Books, Washington, D. C. 1965.

25. Stone, H. S. and A. J. Korenjak. Canonical form and synthesis of cellular cascades. IEEE Transactions on Electronic Computers EC-14:852-862. 1965.

26. Wilcox, R. H. and W. C. Mann. (Eds) Redundancy techniques for computing systems. Washington, D. C., Spartan, 1962.

27. Yoeli, M. Cascade-parallel decompositions of sequential machines. IRE Transactions on Electronic Computers EC-12: 322-324. 1963.

28. Yoeli, M. A group theoretical approach to two-rail cascades. IEEE Transactions on Electronic Computers EC-14:815-822.

29. Zeiger, H. P. Cascade synthesis of finite-state machines. In: IEEE Conference Recordings on Switching Circuit Theory and Logical Design, Ann Arbor, Michigan, 1965. Ann Arbor, 1965. p. 45-51.

APPENDICES

APPENDIX I

List of example machines.

1. M1:

	i_1	i_2	Z
A	C	B	1
B	D	D	0
C	D	C	1
D	C	A	0

2. M2:

	i_1	i_2	Z
A	A	D	0
B	C	D	1
C	E	B	1
D	C	D	0
E	A	B	1

3. M3:

	i_1	i_2	Z
A	C	B	0
B	D	D	0
C	D	C	1
D	C	A	1

4. M4:

	i_1	i_2	Z
A	A	B	0
B	B	A	1

5. M5:

	i_1	i_2	Z
A	C	D	1
B	E	D	0
C	C	A	1
D	C	D	0
E	A	C	0

6. M6:

	$\overline{Y}$	Y	Z
A	B	D	0
B	D	B	1
C	D	D	1
D	B	B	0

APPENDIX II

Some of the better known two dimensional cellular arrays are discussed in this appendix. The material presented here is based on [19].

Spandorfer and his associates at UNIVAC developed several cellular array structures using NAND gates. One of such arrays uses two-input NAND gates. These NAND gates are arranged in two dimensional form as shown in Figure A-1. Two types of "wiggle busses" are used to form the initial connections of the array--the horizontal interconnection structure, called the "horizontal wiggle busses", and the vertical connection structure, called the "vertical wiggle busses". These are also shown in Figure A-1. These wiggle busses are common for any combinational functions having a given number of variables and product terms. The array can be used to realize a particular function by depositing "synthesizing arcs". This is shown in Figure A-2. To synthesize a particular switching function, the function is first expressed in minimal sum-of-products form. Each product term is realized by two columns of NAND gates. Products are then collected (summed) by top two rows of NAND gates. As as example, the realization of the function

$$f = \overline{x}_1 x_3 + x_2 \overline{x}_3 + x_1 x_2 x_3$$

using two-input NAND gate array is shown in Figure A-3. In general, for an n variable function with p product term, $2(n+1)(2p+1)$ cells are required.

Another way of arranging NAND gates in array form is as shown in Figure A-4. This is called the diamond array. In this array, some of the cells are three-input NAND gates. The realization of the same three variable function using diamond array is shown in Figure A-4. The number of cells required is generally less than the one above. For an n-variable function with p product terms, the required cell number is $(n+1)(2p+1)-p$.

Short at SRI developed a type of cellular array using majority gates. It is obvious that a majority gate becomes an AND gate if one of the input is a constant 0 and becomes an OR gates if one of the inputs is a constant 1. A majority-gate array for three input variables is shown in Figure A-5. The function to be realized is first expressed in fundamental sum-of-products form. If the function includes the term $x_1\bar{x}_2x_3$, then the values of f(101) is 1, otherwise is 0. This type of array takes $(2n-1)(2^n+n)$ majority gates for an n variable function.

In a cutpoint cellular array, each cell is able to perform one of the nine specified functions. These functions are listed in Figure A-6 where x and y are cell inputs and z is the cell output. The number N in each cell indicates the function the cell is to perform.

Originally, a specific function is achieved by cutting certain circuit connections within the cell, hence the name cutpoint. Figure A-7 shows the cutpoint cellular array synthesis of the three variable function given earlier. In general, it takes $(n+1)p$ cells for a function of n variables with p product terms.

There are many other types of cellular structures, but those presented above are, in our opinion, more practical and easier to implement.

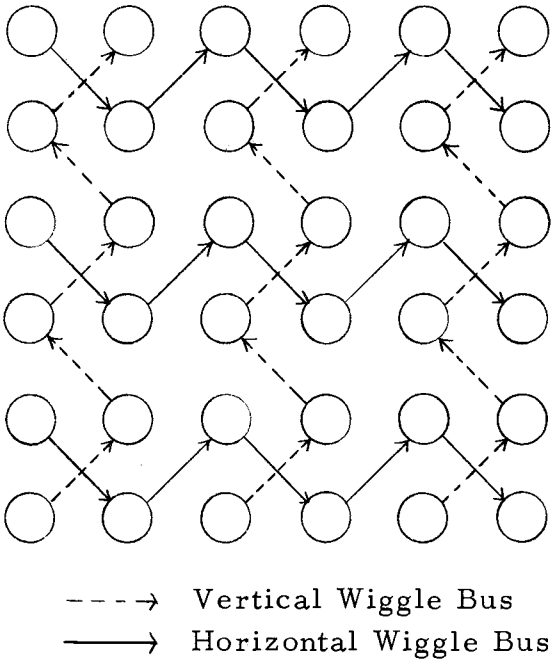


Figure A-1. Two-input NAND gate array.

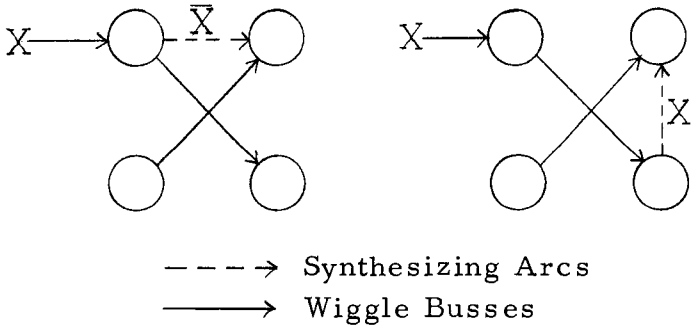


Figure A-2. Synthesizing arcs for two-input NAND gate array.

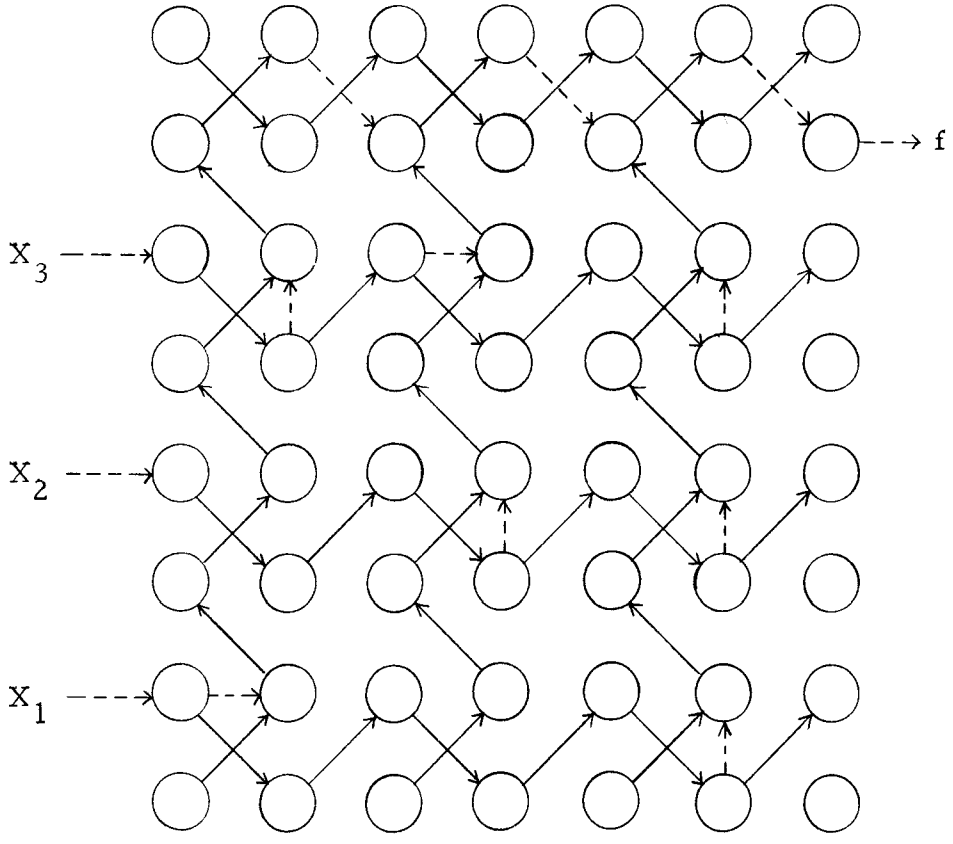


Figure A-3. Cellular synthesis of the function $f = \overline{X}_1 X_3 + X_2 \overline{X}_3 + X_1 X_2 X_3$ using two-input NAND gate array.

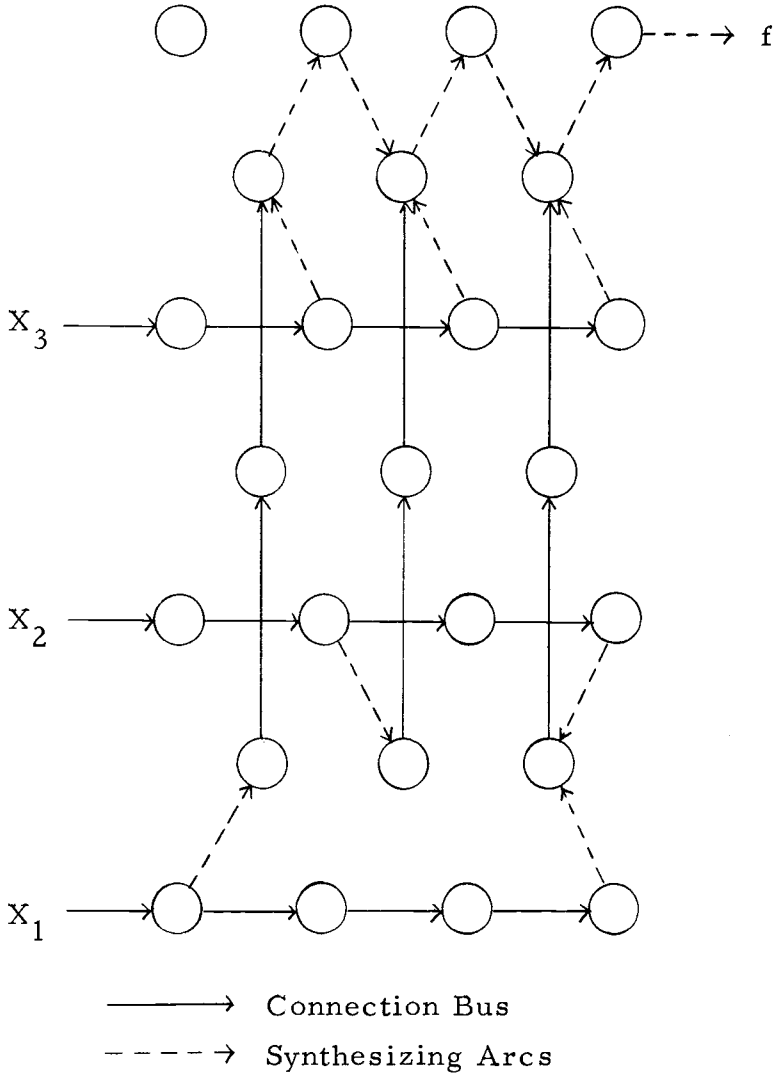


Figure A-4. Diamond array synthesis of the function $f = \bar{X}_1 X_3 + X_2 \bar{X}_3 + X_1 X_2 X_3$.

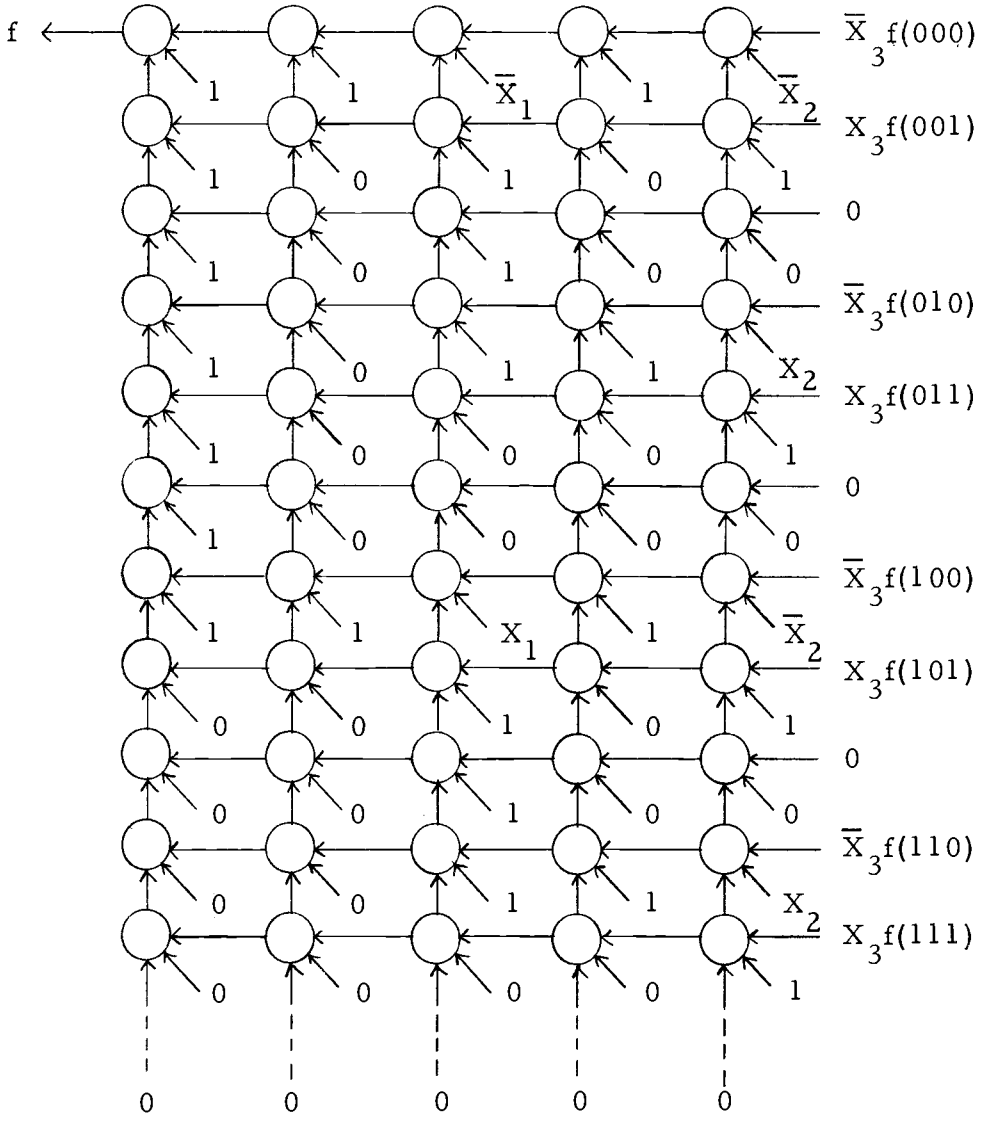
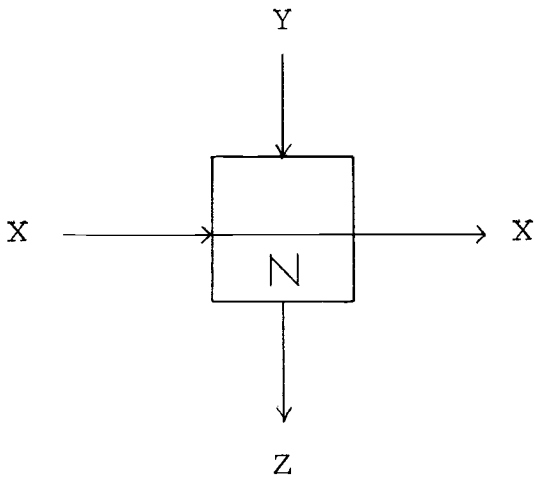


Figure A-5. Majority-gate array for three variable functions.



N	Z
0	1
1	$\overline{Y}$
2	$\overline{X} + \overline{Y}$
3	$\overline{X}\,\overline{Y}$
4	$X + Y$
5	$X\overline{Y}$
6	$X \oplus Y$
7	0
F	X=S, Y=R

Figure A-6. Functional description of a cutpoint cell.

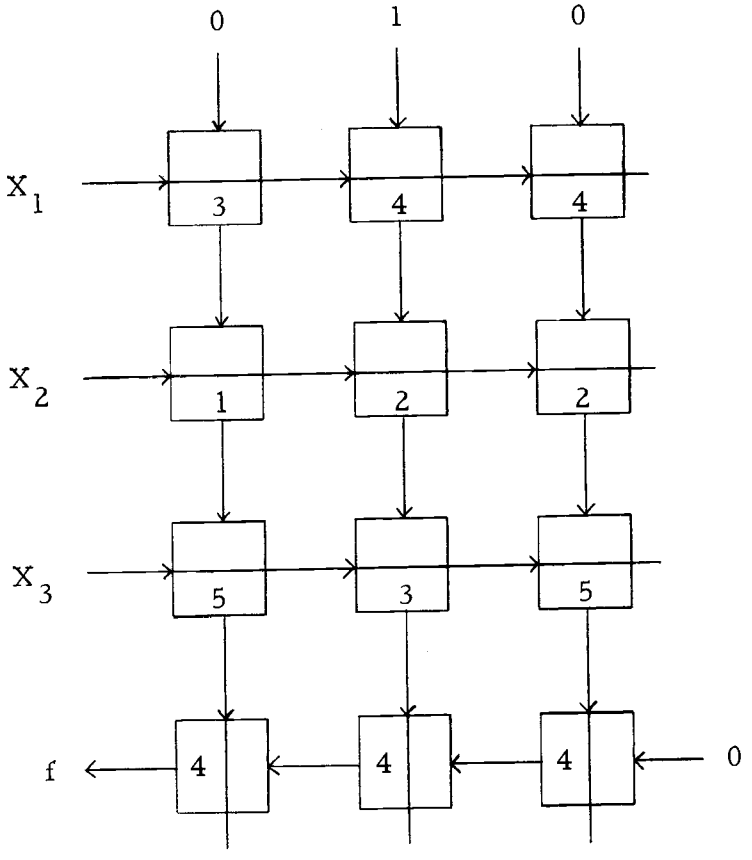


Figure A-7. Cutpoint cellular array synthesis of the function $f = \bar{X}_1 X_3 + X_2 \bar{X}_3 + X_1 X_2 X_3$.