Part 3 Finite-state machines

CHAPTER

9 Introduction to synchronous sequential circuits and iterative networks

In Part 2 we considered combinational switching circuits in which the output values are functions of only the current circuit input values. In most digital systems, however, additional circuits are necessary that are capable of storing information and data and also of performing some logical or mathematical operations upon this data. The output values of these circuits at any given time are functions of external input values as well as of the stored information at that time. Such circuits are called *sequential circuits.*[1]

A *finite-state machine* (or *finite automaton*) is an abstract model describing the synchronous sequential machine and its spatial counterpart, the iterative network. It is the basis for the understanding and development of the various computation structures discussed in Part 3 of this book. The behavior, capabilities, limitations, and structure of finite-state machines are studied in Chapters 12 through 16, while Chapters 9 and 10 are devoted to the synthesis of these machines. Chapter 11 is concerned with asynchronous sequential circuits.

9.1 Sequential circuits – introductory example

In our daily activities, we all encounter the use of various sequential circuits. The elevator control which "remembers" to let us out before it picks up people coming into elevator; traffic-light systems on our roads, trains, and subways; the lock on a safe that not only remembers the combination numbers but also their sequence; all these are examples of sequential circuits in action. Before deriving the basic model and general synthesis procedures, we shall investigate the properties of a simple sequential circuit.

[1] Conventionally, the term *sequential machine* refers to the abstract model that represents the actual *sequential circuit*. In many cases, however, these terms are used interchangeably.

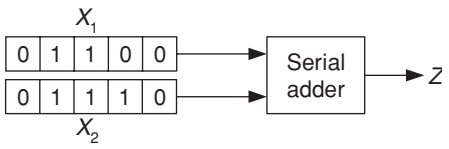


Fig. 9.1 Block diagram of a serial binary adder.

The state table

Consider the *serial binary adder* whose block diagram is shown in Fig. 9.1. It is a synchronous circuit with two inputs, X_1 and X_2, carrying the two binary numbers to be added and one output, Z, which represents the sum. Fixed-length sequences of 0's and 1's are fed to the inputs and obtained at the outputs. The addition is to be performed serially: the least significant digits of numbers X_1 and X_2 arrive at the corresponding input terminals at time t_1; a unit time later, the next-to-least significant digits arrive at the input terminals; and so on. The time interval between the arrival of two consecutive input digits is determined by the frequency of the circuit's clock. We shall assume that the delay within the combinational circuit is small with respect to the clock period (which is the inverse of the clock frequency) and, as a consequence, the sum digit arrives at the Z terminal soon after the arrival of the corresponding input digits at the input terminals.

We shall denote by X and Z the input and output sequences, respectively, and by x and z the input and output symbols at a specified point in time. We may often want to emphasize the precise time at which the input or output value occurs. In such cases, the notation $x(t_i)$, $z(t_i)$ will be used.

Consider the following addition of two binary numbers:

$$\begin{array}{rccccc} & t_5 & t_4 & t_3 & t_2 & t_1 \\ & 0 & 1 & 1 & 0 & 0 = X_1 \\ + & 0 & 1 & 1 & 1 & 0 = X_2 \\ \hline & 1 & 1 & 0 & 1 & 0 = Z \end{array}$$

An examination of the correlation between the input values and the required output value reveals the basic difference between a combinational circuit and the serial binary adder. While in a combinational circuit the output value at time t_i is defined uniquely by the input values at t_i, in the serial adder different output values are required for identical input conditions. For example, at t_1 and t_5 the input values are $x_1x_2 = 00$, but the required output values are $z = 0$ and $z = 1$, respectively. Similarly, at t_3 and t_4 the input values are $x_1x_2 = 11$ while the desired output values are 0 and 1, respectively. It is, therefore, evident that *the output value of the serial adder cannot be specified merely in terms of the external input values*, and so different design procedures must be employed.

Following the rules of elementary binary arithmetic, it is evident that the output value at time t_i is a function of the input values x_1 and x_2 at that time and of the carry that was generated at t_{i-1}. This carry (which may have either

Table 9.1 State table for a serial binary adder

	NS, z			
PS	$x_1x_2 = 00$	01	11	10
A	$A, 0$	$A, 1$	$B, 0$	$A, 1$
B	$A, 1$	$B, 0$	$B, 1$	$B, 0$

of the two values 0 or 1) in turn depends on the input values at t_{i-1} and on the carry generated at t_{i-2}, and so on. Hence the adder must be able to preserve information regarding its input values from the time it is set into operation up to time t_i. However, since the starting time may be long past, it is impossible to preserve the whole history of input values. We therefore seek a different relation between the input values $x_1(t_i)$ and $x_2(t_i)$ and the output value $z(t_i)$, as follows.

In the case of the serial adder, we can distinguish two classes of past input histories, one resulting in the production of a carry 0 and the other in producing a carry 1. These classes will be called the *internal states* (or simply *states*) of the adder. By "memorizing" the value of the carry, the adder actually shows some "trace" of its past input values, at least to the extent of their influence on the response to current input values.

Let A designate the state of the adder at t_i if a carry 0 is generated at t_{i-1}, and let B designate the state of the adder at t_i if a carry 1 is generated at t_{i-1}. We refer to the state of the adder at the time when the current input values are applied to it as its *present state* (*PS*) and the state to which the adder goes, as a result of the new (not necessarily different) carry value, as the *next state* (*NS*). The output value $z(t_i)$ is a function of the input values $x_1(t_i)$ and $x_2(t_i)$ and the state of the adder at time t_i. The next state of the adder depends only on the current input values and on the present state. A convenient way of describing the behavior of the serial adder is by means of a *state table*, as shown in Table 9.1.

Each row of the state table corresponds to a state of the adder, and each column to a particular combination of the external input values x_1 and x_2. Each entry of the table denotes the state to which a transition is made and the output value associated with this transition. For example, if the adder is in state A, i.e., the current carry is 0, and it receives the input combination $x_1x_2 = 11$ then it will go to state B, which corresponds to carry 1, and produce an output value $z = 0$. The remaining entries of the table can be verified in a straightforward manner and, since the table contains eight entries, corresponding to the eight combinations of states and input values, it completely specifies the serial adder.

It is often convenient to use a directed graph as a counterpart to the state table. Such a graph, shown in Fig. 9.2, is known as the *state diagram* (or *state graph*). The vertices and directed arcs of the graph correspond to the states of the adder and to its state transitions, respectively. The labels of the directed

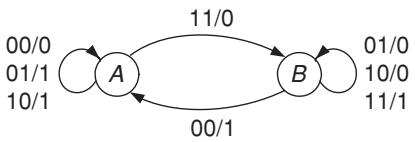


Fig. 9.2 State diagram for a serial adder.

arcs specify the input values and the corresponding output values; e.g., 10/0 represents the condition $x_1 = 1$, $x_2 = 0$, and $z = 0$. Clearly, both the state diagram and state table provide the same information regarding the operation of the adder, and one can be obtained directly from the other. While in many cases these representations are equally suitable, in some applications one may be more convenient than the other.

The state assignment

In order to implement the serial adder, it is necessary to use some device capable of storing the information regarding the presence or absence of a carry. Such a device must have two distinct states, such that each can be assigned to represent a state of the adder. A number of such devices exist, among which is the *delay element*, which may simply consist of a D flip-flop, to be described subsequently. The capability of the delay element to store information is a result of the fact that it takes a finite amount of time for input signal Y to reach its output y. The length of the delay is usually equal to the interval between two successive clock pulses. For convenience, we will assume that this delay is one time unit long.

The state of the delay element is specified by the value of its output y, which may assume either of two values, namely, $y = 0$ or $y = 1$. Since the current input value Y of the delay is equal to its next output value, the input value is referred to as the *next state* of the delay, that is, $Y(t) = y(t + 1)$.

If we assign the states of the delay to those of the adder in such a way that $y = 0$ is assigned to A and $y = 1$ to B, the value of y at t_i will correspond to the value of the carry generated at t_{i-1}. The process of assigning the states of a physical device to the states of the serial adder is known as *state assignment* (or *secondary state assignment*). The output value y is referred to as the *state variable* (or *secondary variable*, to distinguish it from the external primary input variables).

The state assignment is completed by modifying the entries of the state table to correspond to the states of y, in accordance with the selected state assignment. The resulting table is given in Table 9.2, where the next-state and output entries have been separated into two sections. The entries of the next-state table define the necessary state transitions of the adder and thus specify the next value of the output, $y(t + 1)$, of the delay. In addition, since $Y(t) = y(t + 1)$, these entries also specify the input values to the delay at time t required to achieve the

Table 9.2 The transition and output tables for a serial binary adder

	Next state Y				Output z			
	x_1x_2				x_1x_2			
y	00	01	11	10	00	01	11	10
0	0	0	1	0	0	1	0	1
1	0	1	1	1	1	0	1	0

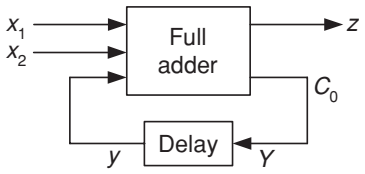


Fig. 9.3 Serial binary adder.

desired state transitions. Thus, the next-state part of Table 9.2, which is called the *transition table*, serves also to specify the required *excitation* of the delay.

The output part of Table 9.2, which is identical to that of Table 9.1, specifies the output value z for every combination of x_1, x_2, and y. Consequently, using the map method the following logic equations result:

$$Y = x_1x_2 + x_1y + x_2y,$$
$$z = x_1'x_2'y + x_1'x_2y' + x_1x_2'y' + x_1x_2y.$$

These equations are clearly identical to those obtained in Section 5.4 for the carry and sum functions of the full adder. The addition is accomplished by retransmitting the carry C_0 of the full adder through the delay Y into the full adder's input, as shown in Fig. 9.3. (Note that a delay whose input is Y is generally referred to as "delay Y.")

9.2 The finite-state model – basic definitions

The behavior of a finite-state machine is described as a sequence of events that occur at discrete instants designated $t = 1, 2, 3$, etc. Suppose that a machine M has been receiving input signals and has been responding by producing output signals. If now, at time t, we were to apply an input signal $x(t)$ to M then its response $z(t)$ would depend on $x(t)$ as well as on the past input signals to M. Also, since a given machine M might have an infinite variety of possible histories, it would need an infinite capacity for storing them.

Since it is impossible to implement machines that have infinite storage capabilities, *we shall concentrate on those machines whose past histories can affect their future behavior in only a finite number of ways*. For example, suppose

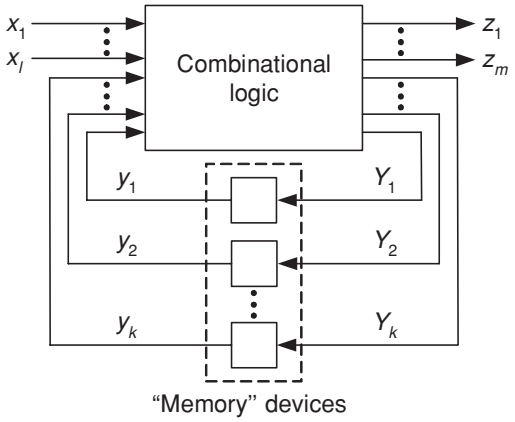


Fig. 9.4 Circuit representation of a synchronous sequential machine.

that the serial binary adder of the previous section has been receiving input signals; its response to the signals at t is a function only of these signals and the value of the carry generated at $t - 1$. Thus, although the adder may have a large number of possible input histories, they may be grouped into two classes, those resulting in a carry 1 and those resulting in a carry 0 at t.

We shall study machines that can distinguish among a *finite* number of classes of input histories and shall refer to these classes as the *internal states* of the machine. Every finite-state machine, therefore, contains a finite number of memory devices, which store the information regarding the past input history. Note that, although we are restricting our attention to machines that have finite storage capacity, no bound has been set on the duration for which a particular input value may affect the future behavior of the machine. A discussion of this subject is deferred to Chapter 14.

Synchronous sequential machines

In general, a synchronous sequential machine is represented schematically by the circuit of Fig. 9.4. The circuit has a finite number l of input terminals. The signals entering the circuit via these terminals constitute a set $\{x_1, x_2, \ldots, x_l\}$ of *input variables*, where each x_j, for all j, may take on one of the two possible values 0 or 1. An ordered l-tuple of 0's and 1's is an *input configuration* (alternatively, *input symbol, pattern*, or *vector*). The set I of $p = 2^l$ distinct input patterns is called the *input alphabet*, and each configuration is referred to as a symbol of the alphabet. Thus, the input alphabet is given by

$$I = \{I_1, I_2, \ldots, I_p\}.$$

For example, if a machine has two input variables x_1 and x_2 then its input alphabet I consists of four symbols (or configurations), that is, $I = \{00, 01, 11, 10\}$.

Similarly, the circuit has a finite number m of output terminals which define the set $\{z_1, z_2, \ldots, z_m\}$ of *output variables*, where each z_j, for all j, is a

binary variable. An ordered m-tuple of 0's and 1's is an *output configuration* (alternatively, *output symbol, pattern*, or *vector*). The set O of $q = 2^m$ ordered m-tuples is called the *output alphabet* and is given by

$$O = \{O_1, O_2, \ldots, O_q\}$$

where each output configuration is a *symbol* of the output alphabet.

The signal value at the output of each memory element is referred to as the *state variable*, and $\{y_1, y_2, \ldots, y_k\}$ constitutes the set of state variables. The combination of values at the outputs of the k memory elements $y_1, y_2, \ldots, y_k$ defines the *present internal state* (or *state*) of the machine. The set S of $n = 2^k$ k-tuples constitutes the entire set of states of the machine, where

$$S = \{S_1, S_2, \ldots, S_n\}$$

The external input values $x_1, x_2, \ldots, x_l$ and the values of the state variables $y_1, y_2, \ldots, y_k$ are supplied to the combinational circuit, which in turn produces the output values $z_1, z_2, \ldots, z_m$ and the $Y_1, Y_2, \ldots, Y_k$ values. The values of the Y's, which appear at the outputs of the combinational circuit at time t, are identical to the values of the state variables at $t + 1$ and, therefore, they define the *next state* of the machine, i.e., the state that the machine will assume next.

Synchronization is achieved by means of clock pulses feeding the memory devices.

Specification of machine behavior

The relationships between the input symbol, present state, output symbol, and next state are described by either a *state table* or *state diagram*. A state table has p columns, one for each input symbol, and n rows, one for each state. For each combination of input symbol and present state, the corresponding entry specifies the output symbol that will be generated and the next state to which the machine will go. Although in practice every machine of the type shown in Fig. 9.4 has 2^l input symbols and 2^k states, some of them may be theoretically unnecessary. In other words, theoretically a machine may have any number p of input symbols and n of states. However, in practice, when realizing such a machine the actual circuit will have $l = \lceil \log_2 p \rceil$ input terminals and $k = \lceil \log_2 n \rceil$ memory elements, where $\lceil g \rceil$ is the smallest integer larger than or equal to g.

To each state of the machine there corresponds a vertex in the state diagram (cf. Fig. 9.2). From each vertex emanate p directed arcs, corresponding to the *state transitions* caused by the various input symbols. Each directed arc is labeled by the input symbol that causes the transition and by the output symbol that is to be generated. Since both the state table and state diagram contain the same information, the choice between the two representations is a matter of convenience, as mentioned above. Both have the advantage of being

precise, unambiguous, and thus more suitable for describing the operation of a sequential machine than any verbal description.

The succession of states through which a sequential machine passes, and the output sequence which it produces in response to a known input sequence, are specified uniquely by the state diagram (or table) and the initial state, where by the *initial state* we refer to the state of the machine prior to the application of the input sequence. The state of the machine after the application of the input sequence is called the *final state*.

9.3 Memory elements and their excitation functions

In discussing the basic model for synchronous sequential machines, we showed that a state table (or diagram) completely specifies the behavior of the machine. In order to design a circuit that operates according to the specifications of a given table, it is necessary first to select a number of memory elements, each of which is a device with two distinct states and is capable of storing a binary digit. The states of these elements are next assigned to the states of the machine, a process known as *state assignment*.

A *transition table* is derived from a state table by the replacement of each next-state entry with the corresponding state of memory elements. A transition table thus specifies for every combination of input values and state variables the next state of the memory elements, which is given by $Y_1, Y_2, \ldots, Y_k$. To generate these Y's, the memory elements must be supplied with appropriate input values. The switching functions, which describe the effect of the circuit inputs $x_1, x_2, \ldots, x_l$ and state variables $y_1, y_2, \ldots, y_k$ on the memory-element inputs, are called *excitation functions*. These functions are derived from an *excitation table*, whose entries are the values of the memory-element inputs.

In Section 9.1, we described the delay element as a memory device. Its storage capability is due to the fact that it takes a finite time for the signal to propagate through it. In practice, the most widely used memory element is the *flip-flop*, which is made up of latches.

Set–reset or SR latch

The set-reset (SR) latch has two inputs, S and R, and two outputs, y and y' (often denoted as the 1 and 0 outputs or Q and Q' outputs, respectively). A block diagram representing an SR latch is shown in Fig. 9.5*a*. Such latches are easily implemented with cross-coupled NOR or NAND gates, as shown in Figs. 9.5*b*, 9.5*c*, respectively.

The SR latch has two states, defined by $y = 1$ and $y = 0$. The output y' is the complement of y. The latch possesses the property that it remains in one state indefinitely until it is directed by an input signal to do otherwise. A signal

Table 9.3 Excitation characteristics of the *SR* latch[a]

$y(t)$	$S(t)$	$R(t)$	$y(t+1)$[b]
0	0	0	0
0	0	1	0
0	1	1	?
0	1	0	1
1	1	0	1
1	1	1	?
1	0	1	0
1	0	0	1

[a] $RS = 0$.
[b] $y(t+1) = R'y(t) + S$.

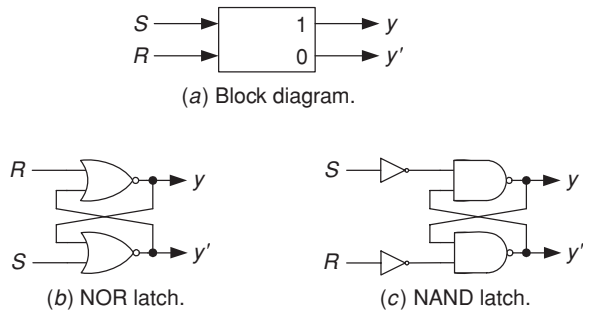


Fig. 9.5 The *SR* latch.

at the input S sets the latch to the 1 state, i.e., it sets $y = 1$; a signal at the input R resets it to the 0 state. The excitation characteristics of the SR latch are given in Table 9.3. If both R and S are excited simultaneously, the operation of the latch becomes unpredictable. Consequently, the requirement that the product $RS = 0$ must be imposed to ensure that the two invalid combinations in Table 9.3 will never occur. The excitation requirements of the SR latch are summarized in Table 9.4, in which a dash denotes a situation where the value of the input is a don't-care, since it does not affect the output value.

In practice, a clocked, or synchronous, version of the SR latch is generally used. In this version, shown in Fig. 9.6, state changes can occur only in synchronization with the pulses from an electronic clock. To ensure proper operation, restrictions must be placed on the length of the clock pulses and on the frequency of the input changes so that the circuit will change state no more than once for each clock pulse. The synchronization of the S and R inputs with the clock is accomplished in Fig. 9.6*b* by AND-gating them before they enter the latch inputs.

Table 9.4 Excitation requirements for the *SR* latch

Change in y		Required value	
from	to	S	R
0	0	0	—
0	1	1	0
1	0	0	1
1	1	—	0

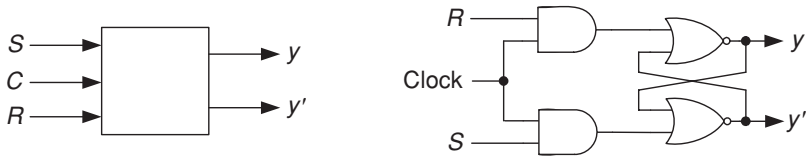


Fig. 9.6 Clocked *SR* latch.

(*a*) Block diagram. (*b*) Logic diagram.

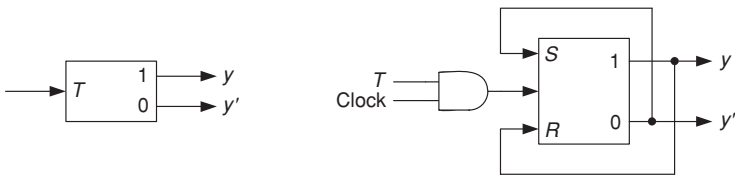


Fig. 9.7 Trigger (or *T*) latch.

(*a*) Block diagram. (*b*) Deriving the *T* latch from the clocked *SR* latch.

To simplify the logic diagrams in subsequent sections we will often ignore the clock, but it is important to note that *in all synchronous circuits, the clock is implicit* whether shown or not.

Trigger or T latch

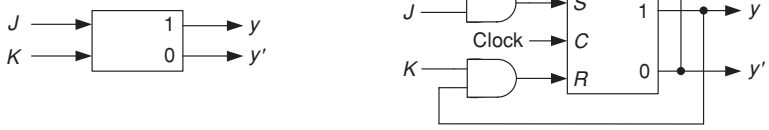
The block diagram of the trigger (T) latch is shown in Fig. 9.7a. The T latch has one input denoted T and two outputs denoted y and y'. It has two distinct states, defined by the logic value of y; namely, the latch is in the 1 state when $y = 1$ and in the 0 state when $y = 0$. The output y' is the complement of y. As in the case of the SR latch, the T latch remains in one state indefinitely until it is directed by an input signal to do otherwise. A value 1 applied to its input triggers the latch and it changes state.

The terminal characteristics of the T latch are summarized in Table 9.5. The next-state function $y(t + 1)$ can be expressed in terms of the present state and

Table 9.5 Excitation requirements for the T latch

Change in y from	to	Required value T
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 9.8 The *JK* latch.



(*a*) Block diagram.

(*b*) Constructing the *JK* latch from the clocked *SR* latch.

input as follows:

$$\begin{aligned} y(t+1) &= Ty'(t) + T'y(t) \\ &= T \oplus y(t). \end{aligned}$$

A clocked T latch can be realized by cross-coupling a clocked SR latch, as shown in Fig. 9.7*b*. (The clock in Fig. 9.6*b* is replaced by an AND combination of the input T and a clock.) If nonclocked operation is desired, the clock and AND gate in Fig. 9.7*b* may be removed and T applied directly to the latch. In the clocked realization, if the value of the output y is 1 then the reset input value is 1. The latch will now change state (to $y = 0$) when $TC = 1$, that is, when the values of T and the clock are both 1. Similarly, when $y = 0$ the set input value is 1, and the latch will change state (to $y = 1$) when $TC = 1$.

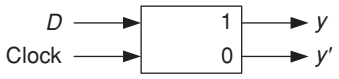
The *JK* latch

The JK latch has the characteristics of both the SR and T latches. Inputs J and K, like S and R, set and reset the latch, respectively. The combination $J = K = 1$ is permitted. When it occurs, the latch acts like a trigger and switches to its complement state; that is, if $y = 1$ it switches to $y = 0$ and vice versa. The block diagram and excitation requirements for the JK latch are shown in Fig. 9.8*a* and Table 9.6, respectively.

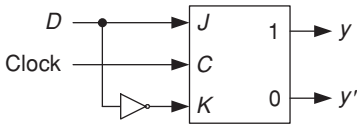
Table 9.6 Excitation requirements for the JK latch

Change in y		Required value	
from	to	J	K
0	0	0	—
0	1	1	—
1	0	—	1
1	1	—	0

Fig. 9.9 The D latch.



(a) Block diagram.



(b) Transforming the JK latch to a D latch.

One possible realization of a clocked JK latch can be obtained by generalizing the clocked SR latch in the way shown in Fig. 9.8b.

The D latch

The block diagram and a possible realization of the D latch are shown in Fig. 9.9. The next state of this device is equal to its present excitation. Hence, it is characterized by the equation

$$y(t+1) = D(t).$$

This latch clearly behaves like the delay element discussed in the preceding sections and, consequently, its excitation requirements are specified by the transition table.

Clock timing and the master–slave flip-flop

A clocked latch is characterized by the fact that it changes states only in synchronization with the clock pulse. Moreover, it changes state only once during each occurrence of a clock pulse. A sequential circuit operating under these restrictions is said to be a *synchronous sequential circuit*. The duration of the clock pulse is usually determined by the circuit delays and signal propagation time through the latches. In fact, *the clock pulse must be long enough to allow the latch to change state and, at the same time, it must be short enough that the latch will not change state twice due to the same excitation.*

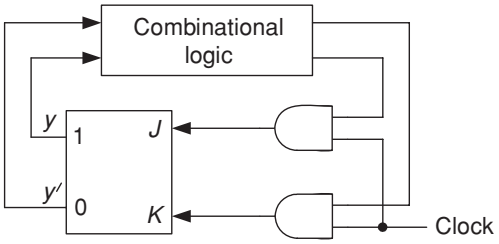


Fig. 9.10 Excitation of a *JK* latch within a sequential circuit.

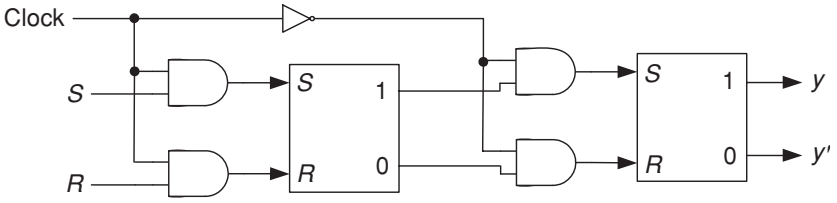


Fig. 9.11 Master–slave *SR* flip-flop.

In general, referring to the sequential circuit model of Fig. 9.4, the outputs of a latch (which serves as a memory element) are inserted into a combinational circuit, which, in turn, generates the excitation functions for that latch, as illustrated in Fig. 9.10. The length of the clock pulse must be such that it will allow the latch to generate the y's but will not be present when the values of the y's have propagated through the combinational circuit. This fine tuning of the length of the clock pulse is difficult to accomplish. To overcome this, another type of synchronous memory element, called a *master–slave flip-flop*, can be used. This flip-flop eliminates the timing problems associated with the feedback loop by essentially isolating the inputs of the flip-flop from its outputs.

A master–slave SR flip-flop, shown in Fig. 9.11, is constructed of two set–reset latches connected in series, with their clock inputs driven in a complementary manner. The first latch, called the *master*, can change state only when the clock is at 1, while the second latch, called the *slave*, can change state only when the clock is at 0. A change in excitation causes a change of state in the master latch. During that period, the slave latch maintains its previous state and serves as a buffer between the master and the next stage. When the clock changes from 1 to 0, the state of the master latch is frozen while the slave latch is enabled and changes its state to that of the master latch. The new state of the slave then determines the state of the entire master–slave flip-flop.

Since the master–slave SR flip-flop still suffers from the drawback that both its inputs cannot simultaneously be 1, it can be converted to a master–slave JK flip-flop to avoid this problem, as shown in Fig. 9.12*a*. Note the similarity to the JK latch shown in Fig. 9.8*b*. The only difference is that the SR latch has been replaced by a master–slave SR flip-flop. Thus, when a master–slave JK flip-flop is substituted for the JK latch in Fig. 9.10, the inputs of the combinational circuit do not change when the clock is at 1. When the clock is at 0, the y's

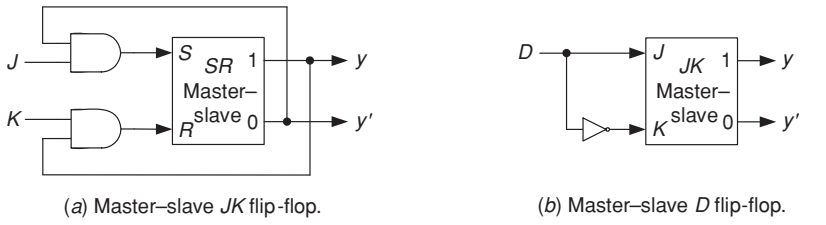


Fig. 9.12 Master–slave flip-flops.

(*a*) Master–slave *JK* flip-flop.

(*b*) Master–slave *D* flip-flop.

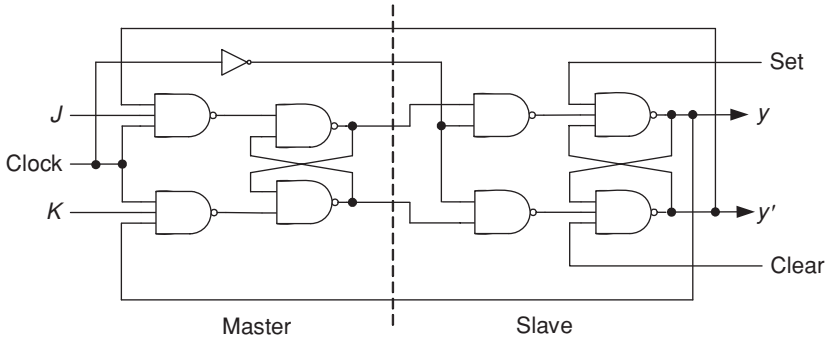


Fig. 9.13 Master–slave *JK* flip-flop with set and clear inputs.

change and, consequently, the output of the combinational circuit changes, but this cannot affect the state of the master latch.

In practice, a master–slave flip-flop has three regular inputs, namely J, K (or S, R) and the clock, and two additional inputs, called (*direct*) *set* and (*direct*) *clear*, as shown in Fig. 9.13. These latter inputs are added to the slave flip-flop and they override the regular input signals and clock. They are used either to set the slave output to 1, by applying 0 to the set input and 1 to the clear input, or to clear the slave output to 0 by applying complementary values to the set and clear inputs. It is not allowable to assign 0's to both the set and clear inputs simultaneously. If we assign both of them 1's, however, the circuit returns to the normal clocked master–slave operation. Such external inputs are very useful, for example, in the design of counters, where it may be necessary to reset a counter to a prespecified count, or in the design of shift registers,[2] which must be cleared before the start of certain computations.

Both master–slave SR and JK flip-flops suffer from the problem of "1's catching" and "0's catching." This arises from the fact that the master latch is transparent when the clock is high. Consider the JK flip-flop shown in Fig. 9.12*a*. When the output of the slave latch is at 0 and the J input has a static-0 hazard (a transient glitch to 1) after the clock has gone high, then the master latch catches this set condition and its output attains the value 1. It then

[2] A shift register consists of a number of cascaded flip-flops.

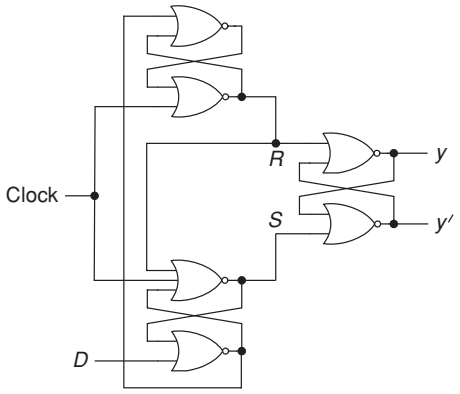


Fig. 9.14 A negative edge-triggered D flip-flop.

passes this 1 to the slave latch when the clock goes low. This leads to "1's catching." Similarly, when the output of the slave latch is at 1 and the K input has a static-0 hazard after the clock has gone high, then the master latch catches this reset condition and its output attains the value 0, which is then passed on to the slave latch when the clock goes low. This leads to "0's catching."

To avoid the above problems, a popular solution is to use a master–slave D flip-flop, as shown in Fig. 9.12*b* (note again the similarity to the D latch shown in Fig. 9.9*b*). Now, even if a static hazard were to occur at the D input when the clock is high, the output of the master latch would revert to its old value when the glitch goes away.

The master–slave T flip-flop can be obtained analogously by replacing the SR latch in Fig. 9.7*b* with master and slave SR latches.

Another type of flip-flop called an *edge-triggered flip-flop* yields a more efficient implementation, in terms of the number of gates, than master–slave flip-flops and hence is popular. This is discussed next.

Edge-triggered flip-flop

A positive (negative) edge-triggered D flip-flop stores the value available on the D input when the clock makes a $0 \rightarrow 1$ ($1 \rightarrow 0$) transition. Any change at the D input after the clock has made a transition does not have any effect on the value stored in the flip-flop.

Consider the negative edge-triggered D flip-flop shown in Fig. 9.14 (a positive edge-triggered flip-flop can be obtained simply by using the complement of the clock). It consists of three latches. When the clock is high, the output of the bottommost (topmost) NOR gate is at D' (D), whereas the S and R inputs of the output latch are both at 0, causing it to hold the previous value. When the clock goes low, the value from the bottommost (topmost) NOR gate gets transferred as D (D') to the S (R) input of the output latch. Thus, the output latch stores the value of D. If there is a change in the value of the D input of the

flip-flop after the clock has made its transition, the output of the bottommost NOR gate attains the value 0 (since its two inputs must have complementary values). However, it can be seen that this cannot change the SR inputs of the output latch.

The excitation characteristics and requirements presented earlier for the various types of latch are also applicable to the corresponding flip-flops. In the subsequent discussion, we shall synthesize sequential circuits using flip-flops. To simplify the resulting tables and circuits, the clock is generally not shown. However, as mentioned before, it is implicit in all synchronous circuits.

9.4 Synthesis of synchronous sequential circuits

We have seen a synthesis procedure in Section 9.1 for a serial binary adder using a delay as the memory element. In this section, we shall develop a general method for designing sequential circuits, using various types of memory elements, and apply this method to the design of some commonly used circuits. The main steps in the method are summarized as follows.

1. From a word description of the problem, form a state table (or a state diagram) that specifies the circuit behavior.
2. Check this table to determine whether it contains any redundant states. (The notion of a redundant state will be defined in Chapter 10, where we shall also present methods for detecting and eliminating such states. The state tables in this section do not contain any redundant states.)
3. Select a state assignment and determine the type of memory elements to be used.
4. Derive the transition and output tables.
5. Derive an excitation table and obtain the excitation and output functions from their respective tables.
6. Draw a circuit diagram.

In effect, in step 5 we are converting a less familiar problem, that of sequential circuit synthesis, into a more familiar problem, that of combinational circuit synthesis, since the construction of the excitation table is actually equivalent to the construction of a set of maps, from which the derivation of the excitation functions is straightforward.

The sequence detector

We wish to design a one-input one-output sequence detector that produces an output value 1 every time the sequence 0101 is detected and an output value 0 at all other times. For example, if the input sequence is 010101 then the corresponding output sequence is 000101. In designing the sequence detector,

Table 9.7 State table for a sequence detector

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 0$	$A, 0$
B	$B, 0$	$C, 0$
C	$D, 0$	$A, 0$
D	$B, 0$	$C, 1$

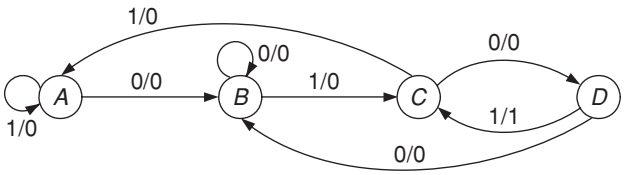


Fig. 9.15 State diagram for a sequence (0101) detector.

we may find it more convenient to start the synthesis procedure by constructing the state diagram of the machine.

At time t_1 the machine is assumed to be in the initial state, designated (arbitrarily) as A. While in this state, the machine can receive input values 0 or 1. For each of these input values, an arc is drawn originating in state A and terminating in the appropriate next state, as shown in Fig. 9.15. The arc labeled 1/0 forms a self-loop around state A, since the machine does not initiate the sequence detection process until it receives a 0 input value. The input value 0 indicates a possible start of the sequence to be detected and, therefore, an arc labeled 0/0 leads from state A to B. When the machine is in state B, a 1 input value takes it to state C, while a 0 input value leaves it in the same state. If, when the machine is in state C, it receives a 1 input value, its last two input values are 11 and, since this input sequence cannot be completed in any way to yield 0101, the machine is directed back to its initial state. The machine arrives at state D after having received an input sequence whose last three symbols are 010. An additional 1 input value produces a 1 output value and causes a transition from state D to C, which is the state corresponding to input sequences whose last two symbols are 01. A 0 input value, applied to the machine when in state D, causes a transition to B because the last 0 symbol may be the prefix of 0101.

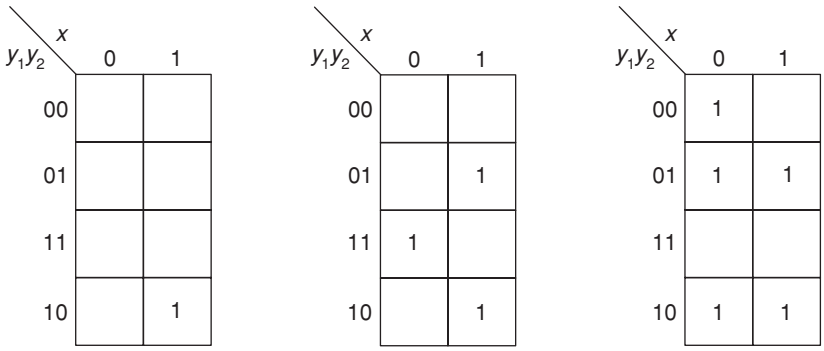
The state table corresponding to the diagram of Fig. 9.15 is given in Table 9.7. The input and output symbols are denoted by x and z, respectively.

Two state variables with $2^2 = 4$ states are needed for the representation of the four states of the sequence detector. If we select two delay elements, Y_1 and Y_2, as memory devices and choose the state assignment shown in the left-hand block of Table 9.8, we obtain the transition and output tables in the center and

Table 9.8 Transition and output tables

		Y_1Y_2		z	
	y_1y_2	$x=0$	$x=1$	$x=0$	$x=1$
A	00	01	00	0	0
B	01	01	11	0	0
C	11	10	00	0	0
D	10	01	11	0	1

Fig. 9.16 Output and excitation maps.



(*a*) z map. (*b*) Y_1 map. (*c*) Y_2 map.

right-hand blocks of Table 9.8. The entries of the transition table specify, for each combination of present state and input symbol, the values that the outputs of the delays should assume next. However, since the next values of the delays are equal to their present excitation, the transition table entries in effect specify the required excitation of the delay elements. Consequently, whenever delay elements are used as memory devices the transition and excitation tables are identical.

The output table is, actually, a three-variable map in which the value of z is specified for every combination of x, y_1, and y_2, as shown in Fig. 9.16*a*. The excitation table consists of two distinct three-variable maps, corresponding to the excitation functions for Y_1 and Y_2. Entries for the map of Y_1 (Y_2) are given by the left-hand (right-hand) entries of the second block of Table 9.8. The logic equations, derived from the maps of Fig. 9.16, for the output and excitation functions are

$$z = xy_1y_2',$$
$$Y_1 = x'y_1y_2 + xy_1'y_2 + xy_1y_2',$$
$$Y_2 = y_1y_2' + x'y_1' + y_1'y_2.$$

The implementation of these equations yields the sequence detector shown in Fig. 9.17.

The reader may have observed that the state assignment employed in Table 9.8 is not the only possible one. In general, different state assignments

Table 9.9 A second assignment

		Y_1Y_2		z	
	y_1y_2	$x=0$	$x=1$	$x=0$	$x=1$
A	00	01	00	0	0
B	01	01	10	0	0
C	10	11	00	0	0
D	11	01	10	0	1

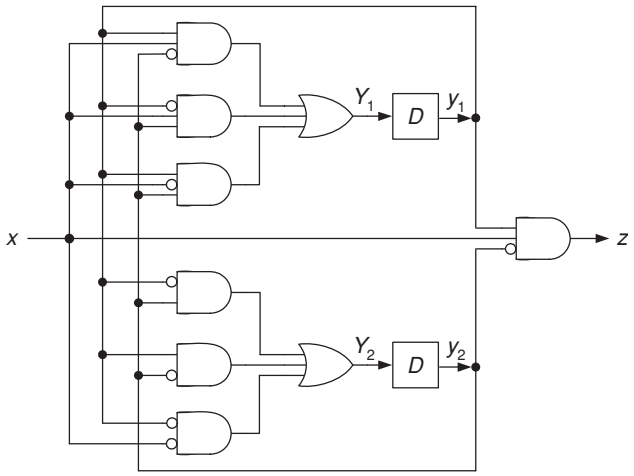


Fig. 9.17 Logic diagram of a sequence detector.

yield different logic equations, which can affect to a considerable degree the area and structure of the resulting circuit. For example, if we interchange the codes assigned to states C and D then we obtain Table 9.9 and the following logic equations:

$$\begin{aligned} Y_1 &= x'y_1y_2' + xy_2, \\ Y_2 &= x', \\ z &= xy_1y_2. \end{aligned}$$

The implementation of the equations derived from this second state assignment requires less than half the number of gates required for the circuit of Fig. 9.17. Also, the second excitation function for Y_2 is independent of the state variables y_1 and y_2; it depends only on the input. Unfortunately, there is no simple procedure that can be used to arrive at an assignment yielding a minimal circuit under some well-defined cost criterion. Some trial and error is consequently necessary until an acceptable assignment is achieved. The state-assignment problem and, in particular, its effect on the machine structure will be discussed extensively in Chapter 12.

Table 9.10 State table for a modulo-8 binary counter

	NS		Output	
PS	$x = 0$	$x = 1$	$x = 0$	$x = 1$
S_0	S_0	S_1	0	0
S_1	S_1	S_2	0	0
S_2	S_2	S_3	0	0
S_3	S_3	S_4	0	0
S_4	S_4	S_5	0	0
S_5	S_5	S_6	0	0
S_6	S_6	S_7	0	0
S_7	S_7	S_0	0	1

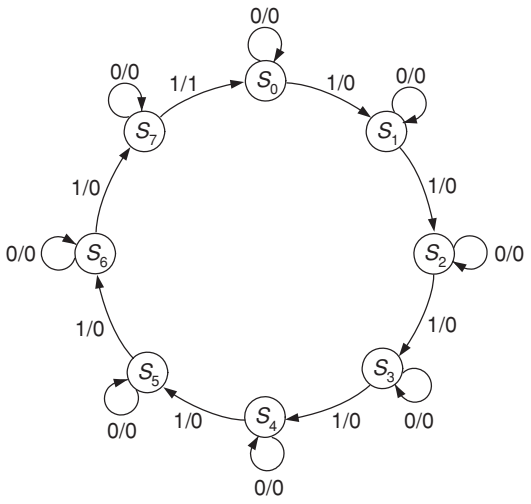


Fig. 9.18 State diagram for a modulo-8 binary counter.

A binary counter

A modulo-8 binary counter is to be designed with one input terminal and one output terminal. It should be capable of counting in the binary number system up to 7 and producing an output value 1 for every eight input 1 values. After a count of seven is reached, the next input value 1 will reset the counter to its initial state, i.e., to a count of zero.

Let $S_0, S_1, \ldots, S_7$ respectively be the states of the counter after it has received $0, 1, \ldots, 7$ input values equal to 1. The state S_0 that designates the zero count is the initial state. Transitions occur between successive states only when the counter receives the input value 1. The state diagram and state table of the counter are shown in Fig. 9.18 and Table 9.10.

Table 9.11 Transition and output tables for a modulo-8 binary counter

PS	NS		z	
$y_3y_2y_1$	$x=0$	$x=1$	$x=0$	$x=1$
000	000	001	0	0
001	001	010	0	0
010	010	011	0	0
011	011	100	0	0
100	100	101	0	0
101	101	110	0	0
110	110	111	0	0
111	111	000	0	1

From the correspondence between the states and the count, it is evident that no state in Table 9.10 is redundant. Also, since the counter has eight states, a state assignment requires three state variables (having $2^3 = 8$ states). The states of these variables, starting from the all-zero position, are 000, 001, ..., 111. The choice of assignment in this example should not be made arbitrarily since it determines the characteristics of the circuits and, in particular, specifies the code and number system in which the counter actually counts. Our objective is to design a counter that counts in the binary number system. Accordingly, the code assigned to each state must be a binary representation of the actual count associated with that state, that is, $S_0 \rightarrow 000$, $S_1 \rightarrow 001, \ldots, S_7 \rightarrow 111$. The transition and output tables corresponding to the foregoing assignment are shown in Table 9.11.

Implementing the counter with *T* flip-flops

To complete the synthesis, we need to choose an appropriate set of memory elements and derive their excitation functions. Let us select T flip-flops whose excitation requirements are specified by Table 9.5.

Up to now we have used a delay element whose output $y(t)$ equals its excitation at time $t-1$ and, consequently, the transition table that specifies the required changes in the values of the y's yields the necessary current excitations as well. Table 9.11, however, does not yield the necessary excitations for the T flip-flops. Consider, for example, entries 000 at the top of the $x=0$ column and the bottom of the $x=1$ column. In the first case the flip-flops remain unchanged, since the transitions are from $S_0 = 000$ to $S_0 = 000$. In the second case, however, the transitions are from $S_7 = 111$ to $S_0 = 000$ and, therefore, all three flip-flops must change state. Hence, while in the first case no excitations are needed, in the second case all three flip-flops must be triggered, i.e., $T_1 = T_2 = T_3 = 1$. Similarly, the transition from $S_5 = 101$ to $S_6 = 110$,

Table 9.12 Excitation table for T flip-flops

	$T_3T_2T_1$	
$y_3y_2y_1$	$x = 0$	$x = 1$
000	000	001
001	000	011
010	000	001
011	000	111
100	000	001
101	000	011
110	000	001
111	000	111

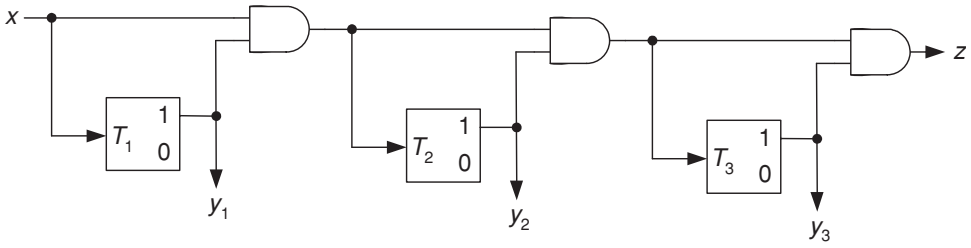


Fig. 9.19 Schematic diagram of a modulo-8 binary counter with T flip-flops.

under $x = 1$, requires y_3 to remain unchanged while y_1 and y_2 change state. Thus, from Table 9.5 it is evident that the required excitation is 011. In the same manner we can specify the required excitations for each transition, and the excitation table shown in Table 9.12 results.

This excitation table consists of three distinct maps specifying T_1, T_2, and T_3 as functions of x, y_1, y_2, and y_3. The logic equations for the output and excitation functions are derived from Tables 9.11 and 9.12, respectively, and are as follows (note that the code resulting from the binary state assignment is not cyclic and thus the reader must be careful when "reading" the equations from the corresponding tables; alternatively, it is possible to transform the tables into three maps and to determine the equations directly from these maps):

$$\begin{aligned} T_1 &= x, \\ T_2 &= xy_1, \\ T_3 &= xy_1y_2, \\ z &= xy_1y_2y_3. \end{aligned}$$

A schematic diagram for a modulo-8 counter is shown in Fig. 9.19. The clock has not been shown but is implicit in this and subsequent figures. A 1 appears on terminal z whenever the total number of 1's received at input line x is a multiple of 8. The actual count (modulo 8) of the number of incoming 1's is given by the values of the state variables y_1, y_2, and y_3, which have binary

Table 9.13 Excitation table for SR flip-flops

	$x = 0$			$x = 1$		
$y_3 y_2 y_1$	$S_3 R_3$	$S_2 R_2$	$S_1 R_1$	$S_3 R_3$	$S_2 R_2$	$S_1 R_1$
000	0–	0–	0–	0–	0–	10
001	0–	0–	–0	0–	10	01
010	0–	–0	0–	0–	–0	10
011	0–	–0	–0	10	01	01
100	–0	0–	0–	–0	0–	10
101	–0	0–	–0	–0	10	01
110	–0	–0	0–	–0	–0	10
111	–0	–0	–0	01	01	01

weights 1, 2, and 4, respectively. For example, if $y_1 = 1$, $y_2 = 0$, and $y_3 = 1$, the number of incoming 1's has been 5 modulo 8, i.e. 5, 13, 21, ...

Implementing the counter with SR flip-flops

The modulo-8 binary counter can also be implemented using SR flip-flops. The excitation table (Table 9.13) is derived from the transition table (Table 9.11) and from the excitation requirements in Table 9.4. As an example, consider the specification of the transition from $S_5 = 101$, under $x = 1$, to $S_6 = 110$. The value of y_1 will change from 1 to 0 and, consequently, the flip-flop must be reset. From Table 9.4, it is evident that this is accomplished by setting $S_1 = 0$ and $R_1 = 1$, and thus the value 01 is entered in row 101, column $S_1 R_1$, of Table 9.13. Similarly, y_2 must change from 0 to 1, and the value 10 is entered in column $S_2 R_2$, row 101. The value of y_3, however, is to remain unchanged; hence R_3 must not be 1 while S_3 may be either 0 or 1, which means that the appropriate entry in row 101, column $S_3 R_3$, is -0. The entire excitation table is specified in a similar way.

Table 9.13 consists of six distinct maps for S_1, R_1, S_2, R_2, S_3, and R_3 as functions of the variables x, y_1, y_2, and y_3. The logic equations for the excitation functions are

$$\begin{aligned} S_1 &= x y_1', \quad & S_2 &= x y_1 y_2', \quad & S_3 &= x y_1 y_2 y_3', \\ R_1 &= x y_1, \quad & R_2 &= x y_1 y_2, \quad & R_3 &= x y_1 y_2 y_3. \end{aligned}$$

The schematic diagram[3] corresponding to these equations is shown in Fig. 9.20.

[3] It is interesting to observe that the binary counter is an iterative network, in the sense that, from the terminal viewpoint, each cell, containing a flip-flop and its associated logic, is indistinguishable from the others. Consequently, in order to design a modulo-16 counter, all that is necessary is to add a fourth identical cell in cascade with the three cells shown in Fig. 9.20.

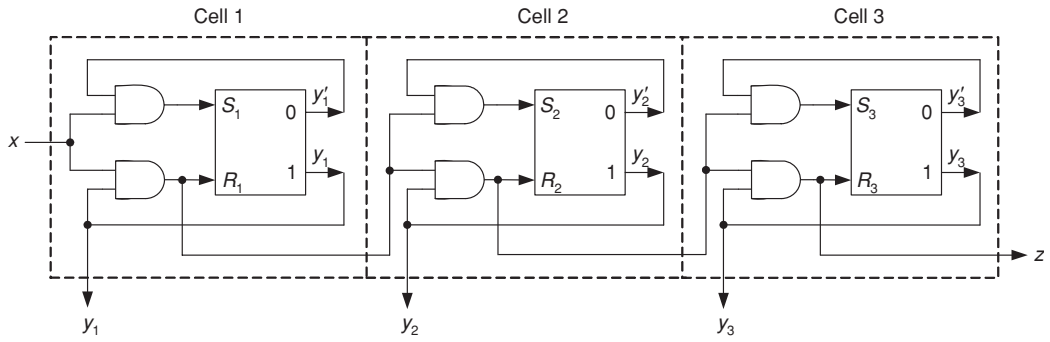


Fig. 9.20 Schematic diagram of a modulo-8 binary counter with *SR* flip-flops.

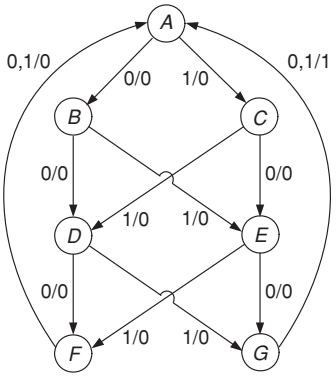


Fig. 9.21 State diagram for a parity-bit generator.

A parity-bit generator

A serial parity-bit generator is a one-terminal circuit that receives coded messages and adds a parity bit to every m-bit message, so that the resulting outcome is an error-detecting coded message. The input values in our example are assumed to arrive in strings of three symbols, i.e., $m = 3$, the strings being spaced apart by single time units. The parity bits are to be inserted in the appropriate spaces, and the resulting outcome is a continuous string of symbols without spaces. Even parity will be used; that is, a parity bit 1 is to be inserted if and only if the number of 1's in the preceding string of three symbols is odd.

The state diagram for the parity-bit generator is shown in Fig. 9.21. States B, D, and F correspond to even numbers of 1's out of one, two, and three incoming input symbols, respectively. Similarly, states C, E, and G correspond to odd numbers of 1's out of one, two, and three incoming input symbols, respectively. From either state F or state G the machine goes to state A, regardless of the input symbol. (Note that, in fact, the fourth input symbol is a blank, i.e., 0.)

Since the state diagram of Fig. 9.21 contains seven states, three state variables are needed for an assignment. However, since three state variables

Table 9.14 State table for a parity-bit generator

	PS	NS		z	
	$y_1y_2y_3$	$x=0$	$x=1$	$x=0$	$x=1$
A	000	B	C	0	0
B	010	D	E	0	0
C	011	E	D	0	0
D	110	F	G	0	0
E	111	G	F	0	0
F	100	A	A	0	0
G	101	A	A	1	1

have a total of eight states, one of the states will not be assigned and so its entries in the corresponding state table may be considered as don't-cares. We shall defer the study of the properties of incompletely specified machines to Chapter 10, however. The state table and a possible state assignment are shown in Table 9.14. The reader can verify that the following logic equations result if JK flip-flops are used as memory elements:

$$J_1 = y_2, \quad J_2 = y_1', \quad J_3 = xy_1' + xy_2, \quad z = y_2'y_3,$$
$$K_1 = y_2', \quad K_2 = y_1, \quad K_3 = y_2' + x.$$

Since the specification of the problem does not offer any clue as to which assignment to select, it may be chosen arbitrarily. The assignment shown in Table 9.14 has been selected so as to yield "reduced dependency" among the state variables; that is, J_1 and K_1 depend only on the second flip-flop while J_2 and K_2 depend only on the first flip-flop. The method of selecting assignments that result in such circuit properties will be presented in Chapter 12.

A sequential circuit as a control element in a computation

In the preceding examples, each sequential circuit received an input sequence and, in turn, produced an output sequence. This output sequence was the objective of the computation. However, many sequential circuits are used to control more complex computations. Indeed, the data for such computations do not even pass through the controlling circuit and are, therefore, not processed by it. The main role of a sequential circuit in the capacity of a control element is to streamline the computation by providing the appropriate control signals. Such circuits usually have a large number of inputs and outputs and, consequently, more informal design techniques simplify the design process considerably. The following example illustrates a simple computation in which a sequential circuit is the control element.

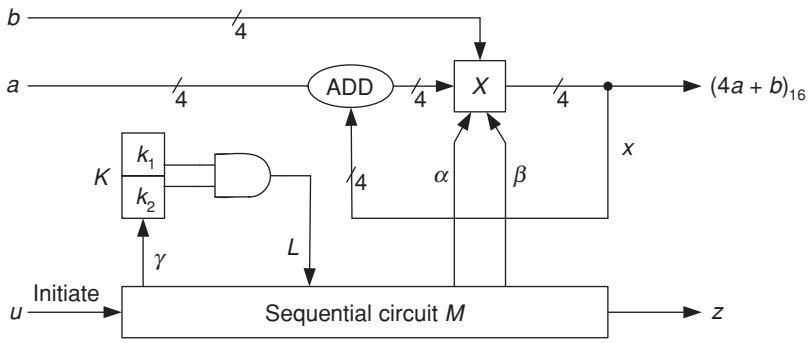


Fig. 9.22 A system to compute $(4a + b)$ modulo 16.

The schematic diagram in Fig. 9.22 describes a digital system that computes the value of $(4a + b)$ modulo 16, where a and b are each a four-bit binary number. In this figure, X is a register[4] containing four flip-flops while x is the number stored in X. The register can be loaded with either b or $a + x$. The addition of a and x is performed by the four-bit parallel adder, denoted ADD. Input b to X is the channel through which the four-bit binary number b is loaded into the register in such a way that each bit enters the corresponding flip-flop. In general, if a number is loaded into the register then it replaces the number presently stored in it. The slash followed by the number 4 across several lines in Fig. 9.22 indicates that each such line actually consists of four wires. The output L of the modulo-4 binary counter K is equal to 1 whenever the count is 3 modulo 4.

The sequential circuit M has two inputs – an input u which initiates the computation and an input L that gives the count of K. It has four outputs, α, β, γ, z, whose tasks are as follows. The outputs α and β are control lines for loading the register X. Whenever $\alpha = 1$, the contents of b are transferred into X. Whenever $\beta = 1$, the values of x and a are added and transferred back into X. The input of the counter is γ. Hence, whenever $\gamma = 1$ the count of K increases by 1. Output z assumes the value 1 whenever the final result is available in X, that is, whenever $x = (4a + b)$ modulo 16. Output z can itself be a control input of another register that is to receive the final result of the computation. However, to simplify the design, this register is not shown.

Initially the count of K is zero, as are the values of u and z. When the value of u becomes 1 the computation starts by setting $\alpha = 1$, which causes b to be loaded into X. Next, a is added to x. This is accomplished by setting β to 1 and, simultaneously, γ to 1, so that the count in K will keep track of the number of times that a has been added to x. After four such additions, z assumes the value 1 and the computation is complete. At this point, the count in K is again zero and, hence, K is ready for the start of the next computation.

[4] A *k-bit register* is a group of k flip-flops such that each flip-flop can store one binary digit and the entire register thus stores a k-bit binary word.

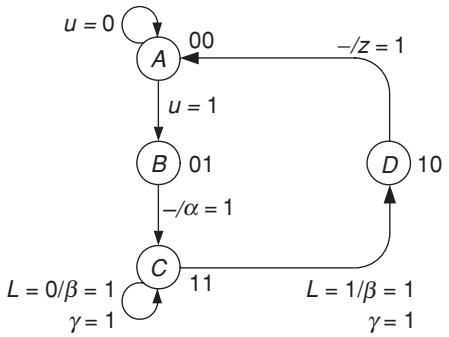


Fig. 9.23 State diagram for circuit M.

A compact state diagram for M is shown in Fig. 9.23. In this diagram, only some of the input and output symbols are shown, in particular, only those that change during the transition and are relevant for the transition in question. The clock is as usual omitted, although it is implicit. Initially, M is in state A. When $u = 1$, M goes to state B without changing the output values. The next clock pulse causes M to go to state C and to produce the output symbol $\alpha = 1$, regardless of the other input symbols. This is indicated by the symbol $-/\alpha = 1$ on the line going from B to C. Register X contains the value of b now. If u is at 1, its value may change to 0 without affecting the computation; u was only needed to cause the transition from A to B and thus initiate the computation. Since $L = 0$, the machine remains in state C and for each clock pulse it produces two output values, $\beta = 1$ and $\gamma = 1$. These output values add a to x while advancing the count in K by one unit. After three such advances, L's value becomes 1 and M goes to state D. During this transition, a is added to x for the fourth time and K is set to zero. At this point, $x = (4a + b)$ modulo 16 and, consequently, z's value becomes 1. The system is now back in state A, ready to start a new computation.

Let the state variables $y_1 y_2$ be assigned to the states of M as follows: $A \rightarrow 00$, $B \rightarrow 01$, $C \rightarrow 11$, $D \rightarrow 10$. This assignment is indicated in Fig. 9.23. The output functions can now be derived directly from the state diagram without any tables or maps. For example, α's value must become 1 whenever the state variable values are $y_1 y_2 = 01$. Thus

$$\alpha = y_1' y_2.$$

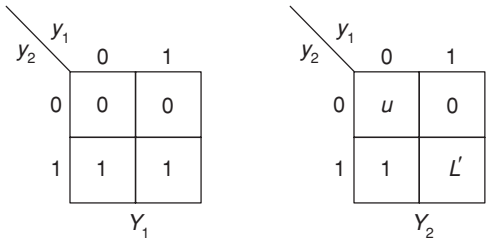
Expressions for the other outputs are obtained in a similar manner:

$$\beta = \gamma = y_1 y_2,$$
$$z = y_1 y_2'.$$

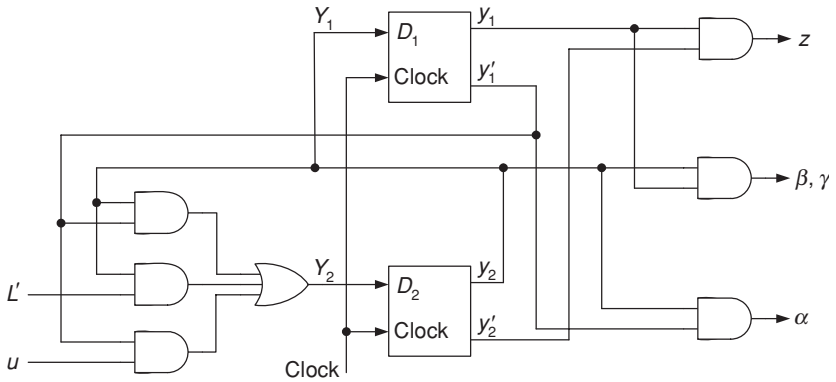
The next-state variables can be obtained with the aid of the transition table shown in Fig. 9.24*a* and the corresponding maps shown in Fig. 9.24*b*, assuming a realization of M by two D flip-flops. In the transition table, some next-state entries are variables, and the treatment of such variables is analogous to the

PS y_1y_2	NS Y_1Y_2
00	$0u$
01	11
11	$1L'$
10	00

(*a*) Transition table.



(*b*) Maps for Y_1 and Y_2.



(*c*) Logic diagram.

Fig. 9.24 Implementing the sequential circuit M with D flip-flops.

treatment of the map-entered variables discussed in Section 4.6. When the present state of M is $y_1y_2 = 00$, the next state depends on u; that is, the next state is 00 if $u = 0$ and 01 if $u = 1$. Consequently, the next-state entry in row 00 is $0u$. However, if the present state is 01 then the next state is 11, regardless of the input values; hence, the next-state entry in row 01 is 11. In a similar manner we derive the entire transition table of Fig. 9.24*a*. The maps in Fig. 9.24*b* are obtained directly from the transition table. For example, the entry in row 11 of the transition table is $Y_1Y_2 = 1L'$. Consequently, a 1 is entered in the Y_1 map in cell 11 while an L' is entered in the same cell in the Y_2 map. Following the procedure for covering maps with map-entered variables, we obtain the following next-state equations:

$$Y_1 = y_2,$$
$$Y_2 = y_1'y_2 + uy_1' + L'y_2.$$

It is useful to note that the next-state equations can also be derived directly from the state diagram: Y_1 is 1 in states C and D, hence it must change to 1 whenever the circuit is in either state B or C. Thus, from the state assignments

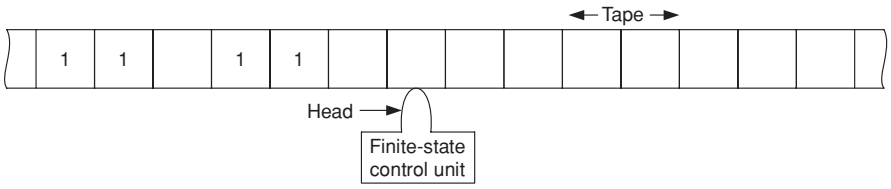


Fig. 9.25 An example of a writing machine.

of these states we obtain

$$Y_1 = y_1' y_2 + y_1 y_2 = y_2.$$

This equation is clearly identical to the one obtained above. Similarly, we can obtain the foregoing equation for Y_2 just by inspecting the state diagram. A logic diagram for M is shown in Fig. 9.24*c*.

9.5 An example of a computing machine

We have been considering sequential machines as independent units possessing finite and limited memory capabilities, whose task is to produce prespecified output sequences in response to the application of external input sequences. Such finite-state machines are known as *nonwriting*, since they have no control on the external input and, in particular, cannot "write" or change their own input symbols. We shall subsequently consider a simple example of a *writing machine*, that is, a finite-state machine that is capable of modifying its own input symbols.

The machine

Consider a system consisting of a finite-state machine M that is coupled through a *head* to an arbitrarily long storage register, called the *tape* (Fig. 9.25). The tape is divided into squares, and each square stores a single symbol at any moment. (Blank squares will be said to store the symbol "blank," denoted 0.) The head is capable of performing three operations, *reading* the symbol contained in the square being scanned, *writing* a new, not necessarily distinct, symbol in the scanned square, and *shifting* the tape one square in either direction. When a new symbol is written on the tape, it replaces the symbol previously there. The finite-state machine acts as the control unit, specifying the operations to be executed by the head. In what is termed a *cycle of computation*, the machine starts in some state S_i, reads the symbol currently being scanned by the head, writes a new symbol there, shifts right or left according to its state table, and then enters state S_j. For convenience, we shall assume that the tape is stationary and the head is moving. Such a machine is usually called a *Turing machine*, after A. M. Turing.

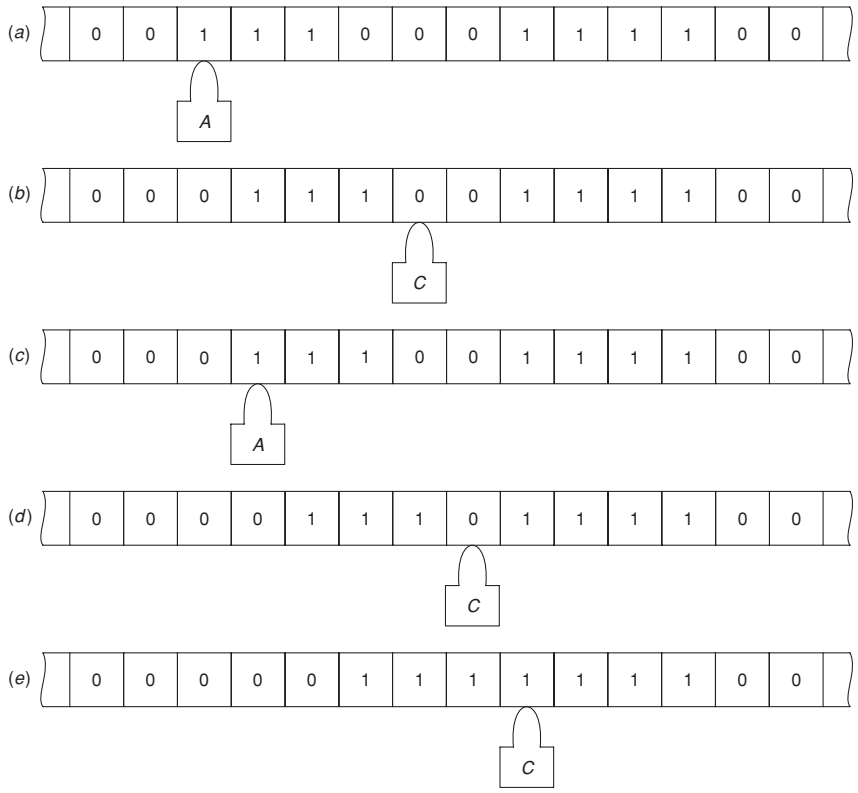


Fig. 9.26 Cycles of computation.

The machine receives its input symbols by reading the pattern of symbols written on the tape. Its output has the dual function of providing the head with the new symbols to be written on the tape and shifting the head in either direction. At the end of the computation, a new pattern of symbols is written on the tape. This pattern is the final objective of the entire computation.

The computation

As an example, let us design a finite-state machine that executes the following computation. The initial pattern of symbols on the tape consists of two finite blocks of 1's separated by a finite block of blanks. The machine is to shift the left-hand block of 1's to the right until it touches the right-hand block, and then halt. The machine is initially in state A, and its head is placed under the leftmost square containing a 1. Let the initial tape consist, for example, of the pattern $\cdots 00111000111100 \cdots$, as shown in Fig. 9.26*a*, where the 0's designate blank squares. The desired final pattern is shown in Fig. 9.26*e*.

A simple way of performing the above computation is to erase, at each step, the leftmost 1 and write a new 1 in the first blank square to the right of the left-hand block of 1's, as shown in Fig. 9.26*b*. This computation is described in

Table 9.15 State table

	NS, write shift	
PS	0	1
A	—	$B, 0R$
B	$C, 1R$	$B, 1R$
C	$D, 0L$	Halt
D	$A, 0R$	$D, 1L$
Halt	Halt	Halt

Table 9.15, where the letters R and L designate right and left shifts, respectively, while 1 and 0 designate the symbols to be written on the tape in each cycle. Thus, for example, the entry $B, 0R$ in row A, column 1, means that the machine is to write symbol 0 in the currently scanned square, shift its head one square to the right, and go to state B.

The computation starts when the machine erases the leftmost 1, currently under the head, shifts one square to the right, and enters state B. As long as it scans squares containing 1 symbols, it leaves them unchanged, shifts to the right, and stays in state B, in accordance with the specification $B, 1R$ in row B, column 1, of the state table. After the third right shift, the head scans a square containing a 0 and, consequently, it must replace it by a 1, shift right, and go to state C. This situation is illustrated in Fig. 9.26*b*.

At this point, the machine is in state C, scanning a 0. The entry in row C, column 0, indicates that the machine is to leave that symbol unchanged, shift left, and enter state D. The machine now moves to the left, leaving all 1's unchanged and remaining in state D until it reaches the first 0 symbol, where it changes direction, shifts right, and enters state A. (See Fig. 9.26*c*.)

The machine is now in a similar situation to that illustrated in Fig. 9.26*a*. Hence, the foregoing sequence of operations will be repeated; that is, the 1 symbol under the head will be replaced by a 0, the machine will move right until it scans the first 0, which it replaces by a 1, shifts right once again, and enters state C. It is now in the position illustrated in Fig. 9.26*d*. The direction of shifts is now to the left until it scans the first 0 symbol, which once again causes a change in the shift direction and sends the machine to state A, with its head scanning the leftmost 1 symbol. After an additional cycle the machine will be in the position shown in Fig. 9.26*e*, in state C and scanning a 1. This terminates the computation, and the machine halts. Clearly, the computation described by Table 9.15 is independent of the precise size of the blocks of 1's and blocks of 0's separating the 1's as long as each block is finite.

The unspecified entry in row A, column 0, is a result of our initial assumption that at the start the head is placed on the leftmost square containing a 1 and, similarly, in all other cases when M enters A it is scanning a 1. This entry may be considered as a don't-care, or alternatively, one may specify that the

machine is to halt, or to cycle in a self-loop, etc. If the initial pattern on the tape contained two or more blocks of 1's, separated by blocks of 0's, the machine will execute the above computation on the two leftmost blocks and will always halt. If, however, it is presented with a tape containing just a single block of 1's then it will shift this block continuously to the right, looking for a second block of 1's, until the entire tape is exhausted. If we assume that the tape is infinite in length, the machine will never halt.

It can be shown that a Turing machine is more powerful than a finite-state machine, in the sense that it can execute computations that cannot be accomplished by any finite-state machine. In the next chapter, we shall show that the preceding computation, for arbitrarily large blocks of 1's, cannot be performed by any finite-state machine. This is clearly a result of the ability of the writing machines to change and write their own input symbols. From a theoretical viewpoint, each finite-state control unit is given access to an arbitrarily large external memory, in which it executes the computations, stores partial results, modifies and replaces input information, and finally stores the output pattern and halts. (We shall keep in mind, however, that there exist computations that never halt, as shown above, but will not refer to them further.)

From the nature of the computations that can be performed by a Turing machine, we may suspect that it can serve as a theoretical model for digital computers. Clearly, no physical computing machine operates as inefficiently as the preceding model, nor does it have an arbitrarily large memory. The model, however, can serve as a tool for studying the capabilities and limitations of physical computing machines, the nature of computations, and the types of function that are not computable by any realizable machine. The study of these important problems is, however, beyond the scope of this book.

Our main objectives in this section have been the introduction of a finite-state machine as the control unit of a larger computing system and the development of a simple model for studying the computation power of digital computers. There is no point in implementing Table 9.15, although this could be accomplished in the usual manner.

9.6 Iterative networks

An *iterative network* is a digital structure composed of a cascade of identical circuits or *cells*. An iterative network may be sequential in nature, where each cell is a sequential circuit, e.g., the counter in Fig. 9.20 or a shift register, or it may be a combinational network where each cell is itself a combinational network. The description and synthesis of combinational iterative networks are similar to those of synchronous sequential circuits. Moreover, it will be shown that every finite output sequence that can be produced *sequentially* by a sequential machine can also be produced *spatially* (or simultaneously) by a combinational iterative network.

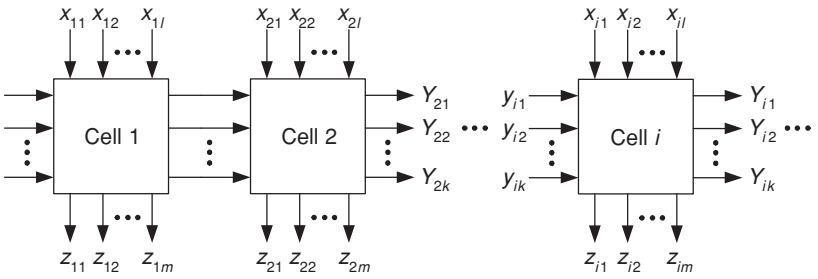


Fig. 9.27 General structure of an iterative network.

Because an iterative network consists of identical cells, we shall restrict our attention to the design of any arbitrary cell, which will be referred to as a *typical cell*.

The analogy between iterative networks and sequential machines

The general structure of an iterative network is shown in Fig. 9.27. The external *cell inputs* applied to the ith cell are designated $x_{i1}, x_{i2}, \ldots, x_{il}$, where the ith (typical) cell is counted from the left. The *cell outputs* are designated $z_{i1}, z_{i2}, \ldots, z_{im}$. In addition, each cell receives information from the preceding cell via the intercell carry wires $y_{i1}, y_{i2}, \ldots, y_{ik}$, which are called *input carries*, and transmits information to the next cell via the intercell carry wires $Y_{i1}, Y_{i2}, \ldots, Y_{ik}$, called *output carries*. Often, we are interested only in the output values from the rightmost cell. In this case the cell outputs are eliminated and the output is taken from the output carries of the last cell.

The operation of a cell can be described by means of a *cell table*, which specifies, for each combination of cell inputs and input carries, the values of the cell outputs and output carries. For example, let us construct the iterative network analogous to the sequence detector of Section 9.4. That is, we want to design an iterative network that consists of an arbitrarily large number of cells and whose typical cell contains a single cell input x_i and a single cell output z_i. The input symbols are applied to all cells *simultaneously* and the output symbols are assumed to be generated *instantaneously* in such a way that the output z_i is 1 if and only if the input pattern of the four cells $i-3, i-2, i-1$, and i is 0101, i.e., $x_{i-3} = x_{i-1} = 0$, and $x_{i-2} = x_i = 1$.

The technique of specifying the cell table for the ith cell is similar to that used in forming Table 9.7. The table must have four rows (or states), corresponding to the four possible distinct signals delivered by the intercell input carries. The resulting table, which is identical to Table 9.7, is repeated in Table 9.16. Row D designates the signals received by the ith cell when the input pattern in the three preceding cells is 010. Similarly, row C designates the signal when the input pattern in the two preceding cells is 01, and so on. From these incoming intercell signals and from cell input x_i, the ith cell can compute the necessary

Table 9.16 Cell table for an iterative pattern detector

	NS, z_i	
PS	$x_i = 0$	$x_i = 1$
A	$B, 0$	$A, 0$
B	$B, 0$	$C, 0$
C	$D, 0$	$A, 0$
D	$B, 0$	$C, 1$

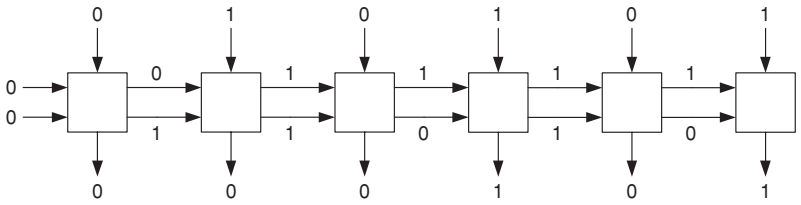


Fig. 9.28 Pattern detection.

cell output value and the signals to be transmitted to the next cell via the output carry wires.

If we specify the intercell signals in such a way that A is represented by $y_{i1}y_{i2} = 00$, B by 01, C by 11, and D by 10, the transition table shown in Table 9.8 results and, as a consequence, the logic equations derived in Section 9.4 are obtained. In general, *if the same assignment is selected for the iterative network as for the sequential circuit, the logic circuit of the ith cell and the combinational logic of the sequential circuit are identical*. While in the sequential case information is fed back through delays, in the iterative network, the entire computation is executed by using many identical cells. Clearly, the number of cells in an iterative network must equal the length of the input patterns applied to it. For example, if the input patterns are limited to length 6, and the specific input pattern applied to the above pattern detector has the form 010101, then the resulting output pattern will be 000101, as shown in Fig. 9.28. (The symbols along the intercell carry leads denote the transmitted signals.)

The reader is encouraged to apply the foregoing procedure and to design a parallel parity-bit generator as a counterpart to the sequential parity-bit generator specified by Table 9.14.

Synthesis

The synthesis procedure for iterative networks is best illustrated by an example. We wish to design an n-cell network where each cell has one cell input x_i and

Table 9.17 Cell table

	NS, z_i	
PS	$x_i = 0$	$x_i = 1$
A	$A, 0$	$B, 1$
B	$B, 1$	$C, 1$
C	$C, 1$	$D, 0$
D	$D, 0$	$D, 0$

Table 9.18 Output carries and cell output table

	$Y_{i1}Y_{i2}, z_i$	
$y_{i1}y_{i2}$	$x_i = 0$	$x_i = 1$
00	00, 0	01, 1
01	01, 1	11, 1
11	11, 1	10, 0
10	10, 0	10, 0

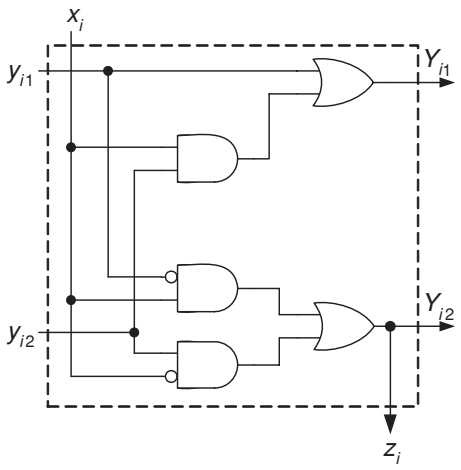


Fig. 9.29 Iterative network cell derived from Table 9.18.

one cell output z_i, such that $z_i = 1$ if and only if either one or two of the cell inputs $x_1, x_2, \ldots, x_i$ have the value 1.

The cell table of the ith cell must have at least four rows to distinguish the following four distinct states. Row A designates the state where none of the cell inputs to preceding cells has the value 1. Similarly, rows B, C, and D designate, respectively, the states where one, two, three or more of the cell inputs to preceding cells have the value 1. The resulting cell table is given as Table 9.17. The state assignment and output tables are shown in Table 9.18, and the typical cell is shown in Fig. 9.29.

The logic equations corresponding to the output carries and the ith cell output are

$$\begin{aligned} Y_{i1} &= y_{i1} + x_i y_{i2}, \\ Y_{i2} &= x_i' y_{i2} + x_i y_{i1}', \\ z_i &= Y_{i2}. \end{aligned}$$

As a consequence of their iterative structure, such networks are easier to design and construct. The time of operation may be substantially longer

than for other possible realizations, however. When realizing combinational circuits, for which the speed of operation is not crucial and which can be composed of identical cells, iterative networks prove to be very useful and economical.

Notes and references

The finite-state model described in this chapter was proposed by Mealy [7] in 1955, on the basis of earlier models by Huffman [3] and Moore [8]. The applicability of the model to iterative combinational circuits was pointed out by McCluskey [6]. Recently, there have been several texts devoted to finite-state machines, among which are Hill and Peterson [2], Katz [4], Mano and Ciletti [5], and Wakerly [10]. A collection of original basic papers dealing with various aspects of finite automata is available in a book edited by Moore [9]. A comprehensive presentation of iterative networks is available in Hennie [1].

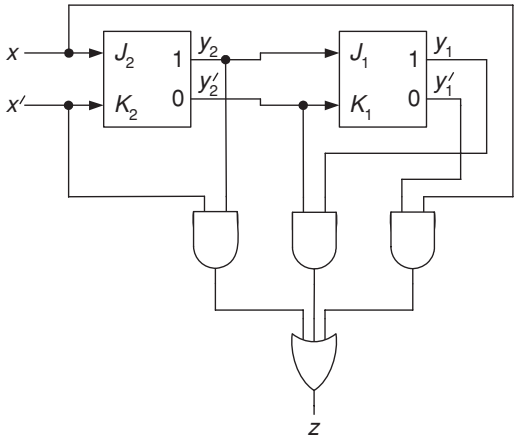
[1] Hennie, F. C.: *Iterative Arrays of Logical Circuits*, MIT Press, Cambridge MA, 1961.
[2] Hill, F. J., and G. R. Peterson: *Computer Aided Logical Design With Emphasis on VLSI*, fourth edition, John Wiley & Sons, New York, 1993.
[3] Huffman, D. A.: "The synthesis of sequential switching circuits," *J. Franklin Inst.*, vol. 257, pp. 161–190, March 1954; pp. 275–303, April 1954. Reprinted in Moore [9].
[4] Katz, R. H., and G. Borriello: *Contemporary Logic Design*, second edition, Pearson Prentice Hall, Upper Saddle River NJ, 2005.
[5] Mano, M. M., and M. D. Ciletti: *Digital Design*, fourth edition, Prentice Hall, Upper Saddle River, NJ, 2007.
[6] McCluskey, E. J.: "Iterative combinational switching networks: general design considerations," *IRE Trans. Electron. Computers*, vol. EC-7, pp. 285–291, December 1958.
[7] Mealy, G. H.: "A method for synthesizing sequential circuits," *Bell System Tech. J.*, vol. 34, pp. 1045–1079, September 1955.
[8] Moore, E. F.: Gedanken-experiments on sequential machines, pp. 129–153, *Automata Studies*, Princeton University Press, 1956.
[9] Moore, E. F. (ed.): *Sequential Machines: Selected Papers*, Addison Wesley, Reading, Mass., 1964.
[10] Wakerly, J. F.: *Digital Design Principles and Practices*, Prentice Hall, Englewood Cliffs NJ, 1990.

Problems

Problem 9.1. Analyze the synchronous circuit of Fig. P9.1 (the clock is not shown, but is implicit).

(a) Write down the excitation and output functions.
(b) Form the excitation and state tables.
(c) Give a word description of the circuit operation.

Fig. P9.1



Problem 9.2. A long input sequence enters a one-input one-output synchronous sequential circuit, that is required to produce an output symbol $z = 1$ whenever the sequence 1111 occurs. Overlapping sequences are accepted; for example, if the input sequence is $01011111\cdots$, the required output sequence is $00000011\cdots$.
(a) Draw a state diagram.
(b) Select an assignment and show the excitation and output tables.
(c) Write down the excitation functions for SR flip-flops, and draw the corresponding logic diagram.

Problem 9.3. Repeat Problem 9.2 for the sequence 01101, and implement the circuit with T flip-flops as memory elements.

Problem 9.4. Construct the state diagram for a one-input eight-state machine that is to produce an output symbol $z = 1$ whenever the last string of five input symbols contains exactly three 1's and starts with two 1's. After each string that starts with two 1's, analysis of the next string does not start until the end of this string of five symbols, whether it produces an output value 1 or not. For example, if the input sequence is 11011010 then the output sequence is 00000000, while an input sequence 10011010 produces an output sequence 00000001.

Problem 9.5. For each of the following cases, show the state table that describes a one-input one-output machine having the following specifications.
(a) An output symbol $z = 1$ is to be produced to coincide with every occurrence of the input symbol 1 following a string of two or three consecutive 0's at the input. At all other times, the output symbol is to be 0.
(b) Regardless of the input symbols, the first two output symbols are 0's. Thereafter, output symbol z is a replica of input symbol x but delayed by two time units, that is, $z(t) = x(t-2)$ for $t \geq 3$.
(c) The output $z(t)$ is 1 if and only if $x(t) = x(t-2)$. At all other times, z is to be 0.

(d) The output z has the value 1 whenever the last four input symbols correspond to a BCD number that is a multiple of 3, i.e., 0, 3, 6, ...

Problem 9.6. Design a one-input one-output synchronous sequential circuit that produces an output symbol $z = 1$ whenever any of the following input sequences occurs: 1100, 1010, or 1001. The circuit resets to its initial state after an output symbol 1 has been generated.
(a) Form the state diagram or table. (Seven states are sufficient.)
(b) Choose an assignment, and show the excitation functions for JK flip-flops.

Problem 9.7. Design a one-input one-output synchronous sequential circuit that examines the input sequence in nonoverlapping strings having three input symbols each and produces an output symbol 1 that is coincident with the last input symbol of the string if and only if the string consisted of either two or three 1's. For example, if the input sequence is 010101110, the required output sequence is 000001001. Use SR flip-flops in your realization.

Problem 9.8. Design a modulo-8 counter that counts in the way specified in Table P9.8. Use JK flip-flops in your realization.

Table P9.8

Decimal	Gray code		
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

Problem 9.9. Construct the state diagram for a synchronous sequential machine that can be used to detect faults in coded messages of the 2-out-of-5 type. That is, the machine examines the messages serially and produces an output symbol 1 whenever an illegal message of five binary digits is detected.

Problem 9.10. When a certain serial binary communication channel is operating correctly, all blocks of 0's are of even length and all blocks of 1's are of odd length. Show the state diagram or table of a machine that will produce an output symbol $z = 1$ whenever a discrepancy from the above pattern is detected. The following is an example.

X:	0	0	1	0	0	0	1	1	1	0	1	1	0	0	$\cdots$
Z:	0	0	0	0	0	0	1	0	0	0	1	0	1	0	$\cdots$

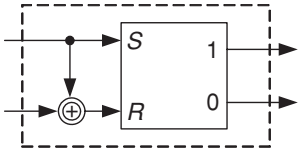
Problem 9.11. A new kind of flip-flop has been designed. It is equivalent to an SR flip-flop with gated inputs, as shown in Fig. P9.11.

A synchronous sequential circuit that generates an output symbol $z = 1$ whenever the string 0101 is scanned in the input sequence is to be designed. Overlapping strings

are accepted; for example, corresponding to the input sequence 0010101, the required output sequence is 0000101.

(a) Construct the state diagram and table for the circuit, using the letters A, B, C, etc.
(b) Make a state assignment (use a Gray code, starting with an all-0 assignment for the initial state).
(c) Realize the sequential circuit using the new flip-flops as memory elements. Give the logic equations for the memory elements and the output.

Fig. P9.11



Problem 9.12. The clocked memory device shown in Fig. P9.12 has one binary input Y and one binary output y. If $Y(t) = 0$ then $y(t + 1) = 0$; if $Y(t) = 1$ then $y(t + 1) = y'(t)$.

(a) The state table given in Table P9.12 is to be realized using two such memory devices. Choose an appropriate state assignment and give the corresponding excitation and output equations.
(b) Briefly discuss the possibility and practicality of using such memory devices to realize an arbitrary state table.

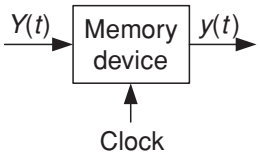


Fig. P9.12

Table P9.12

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 0$	$B, 0$
B	$C, 0$	$A, 1$
C	$B, 0$	$D, 0$
D	$C, 0$	$B, 1$

Problem 9.13. Write the state table for a synchronous circuit, with one input x and one output z, that operates according to the following specifications. At time $t = 0$, the initial state is A, and $x(t) = 0$ for $t < 0$. The output function is given by either (a) or (b) as follows:

(a) $z(t) = x(t) + x(t - 1)$,
(b) $z(t) = x(t) \cdot x(t - 1)$

where the change from (a) to (b) occurs at times τ such that

$$x(\tau) = x(\tau - 1) = x(\tau - 2) = 1$$

and the change from (b) to (a) occurs at times T such that

$$x(T) = x(T - 1) = x(T - 2) = 0.$$

An example is shown in Fig. P9.13.

Fig. P9.13

$t =$	0	1	2	3	4	5	6	7	8	9	10	11	12	…
$x(t) =$	0	1	1	1	0	0	1	1	0	0	0	1	0	…
$z(t) =$	0	1	1	1	0	0	0	1	0	0	0	1	1	…
		(a)				(b)						(a)		

Problem 9.14. The synchronous circuit shown in Fig. P9.14, where D denotes a unit delay, produces a periodic binary output sequence. Assume that initially $x_1 = 1$, $x_2 = 1$, $x_3 = 0$, $x_4 = 0$ and that the initial output sequence is 1100101000. Thereafter, this sequence repeats itself. Find a minimal expression for the combinational circuit $f(x_1, x_2, x_3, x_4)$.

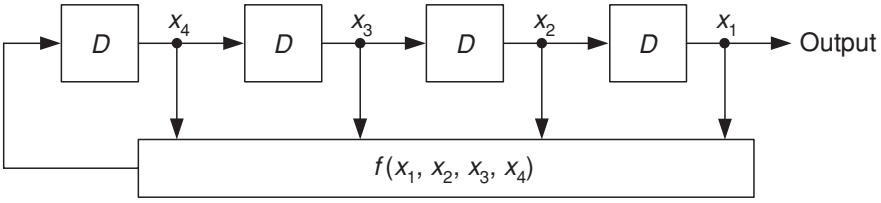


Fig. P9.14

Problem 9.15. A synchronous machine N is part of a transmitter and is used to encode binary serial messages. The coded messages are then transmitted to a receiver, as shown in Fig. P9.l5. The receiver contains a synchronous machine M that is used to decode the received messages.

(a) Given that the initial state of N is A, find the state diagram of machine M.

(b) Suppose the initial state of N is unknown and machine M received a 10-bit message; which of the 10 bits can be uniquely decoded without an error? Explain.

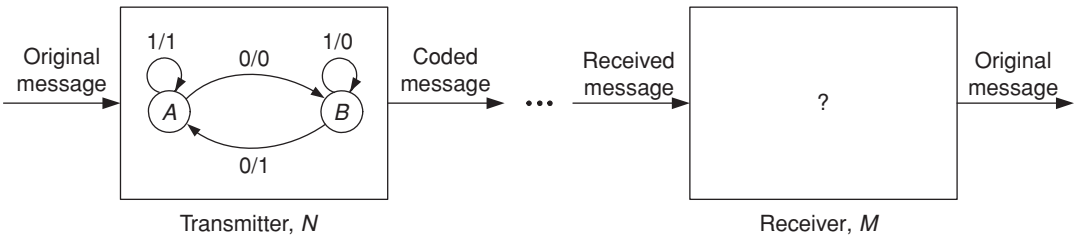


Fig. P9.15

Problem 9.16. A *palindrome* is a sequence which reads the same backward as forward, e.g., 11011 or 01010. Show the finite-state control of a Turing machine that is capable of detecting arbitrarily long palindromes. Assume that you are given a tape initially marked only with symbols #, 0, 1, where the blanks (#) separate blocks of intermixed 0's and 1's. The machine will be started on a # and then checks whether the sequence to its right is a palindrome. If not, the machine should proceed to the next block. If the sequence is a palindrome, the machine should stop at the # to the right of the block. An example is shown in Fig. P9.16.

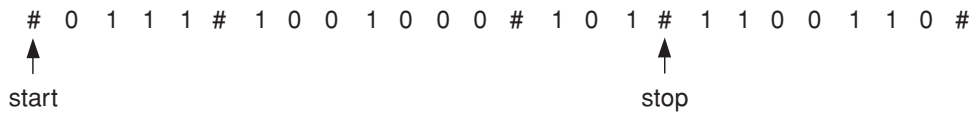


Fig. P9.16

Hint: It is often useful in the course of computation to mark certain digits. This can be accomplished by replacing those digits with different symbols; for example, 0's may be replaced by 2's, while 1's may be replaced by 3's, etc. When these markers are no longer necessary, they are replaced with the old symbols. Use as many new symbols as necessary.

Problem 9.17. Assume that you have a Turing machine that is started at the leftmost 1 in a block of n 1's on a tape that otherwise contains only #'s (blanks), as shown in Fig. P9.17. Using as many symbols as you like:

(a) Show a finite-state control that will duplicate the block of 1's immediately to the right of the original block, leaving the original block and the rest of the tape intact when the machine stops (viz., the block is simply doubled in size – it now contains $2n$ 1's). The machine should stop at the leftmost 1.

(b) Show a finite-state control that will produce a number of replicas equal to the original number of 1's (it stops with a block of n^2 1's).

(c) Show a finite-state control that will increase the number of 1's to 2^n and will then stop.

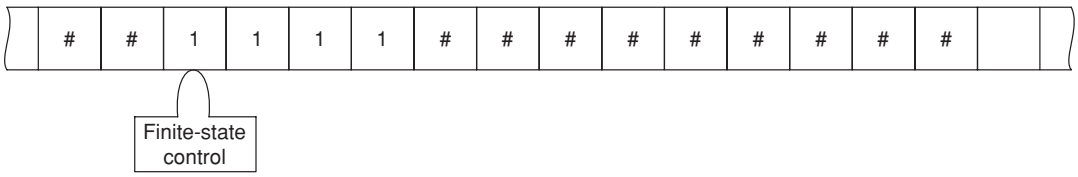


Fig. P9.17

Problem 9.18. An iterative network to be used for detecting faults in Ringtail-coded messages is to be designed. The network consists of five cells, each receiving a digit of the coded message, and is to produce an output symbol 1 when and only when an illegal message is detected. (The Ringtail code is defined in Problem 5.2.)

(a) Construct a cell table.

(b) Select an assignment and derive the logic equations for the output carries and the cell output.

(c) Construct a typical cell using AND, OR, and NOT gates.

Problem 9.19. The cell output of a typical cell of an iterative network has the value 1 if and only if the input pattern of the preceding cells consists of groups of 0's and 1's such that each group contains an odd number of members.

(a) Construct a cell table.

(b) Realize the typical cell using AND, OR, and NOT gates.

Problem 9.20. The typical cell of an iterative network has one binary input x_i and one binary output z_i. The output $z_i = 1$ if and only if $x_i \neq x_{i-2}$. For the first two cells (i.e., $i = 1, 2$), assume that $x_{-1} = x_0 = 0$.
(a) Construct a cell table.
(b) Make a Gray-code state assignment and give the output and carry functions.

CHAPTER

10 Capabilities, minimization, and transformation of sequential machines

This chapter extends some of the concepts introduced in Chapter 9 and presents important techniques for the synthesis of sequential machines and for other problems considered in later chapters. The first two sections are concerned with the general finite-state model, its definition, capabilities, and limitations. The last two sections are concerned with the minimization of completely, as well as incompletely, specified machines.

10.1 The finite-state model – further definitions

Our attention will be focused primarily on *deterministic machines*, which possess the property that the next state $S(t+1)$ is determined uniquely by the present state $S(t)$ and the present input $x(t)$. Thus,

$$S(t+1) = \delta\{S(t), x(t)\}, \tag{10.1}$$

where δ is called the *state transition function*. The value of the output $z(t)$ is, in the most general case, a function of the present state $S(t)$ and the inputs $x(t)$, i.e.,

$$z(t) = \lambda\{S(t), x(t)\}, \tag{10.2}$$

where λ is called the *output function*. A machine possessing properties in Eqs. (10.1) and (10.2) is generally known as a *Mealy machine*. Another machine, known as a *Moore machine*, results when the output is a function of only the present state and is independent of the external input. In this case,

$$z(t) = \lambda\{S(t)\}. \tag{10.3}$$

Thus, we arrive at the following formal definition of a sequential machine.

Definition 10.1 A *synchronous sequential machine M* is a quintuple

$$M = (I, O, S, \delta, \lambda),$$

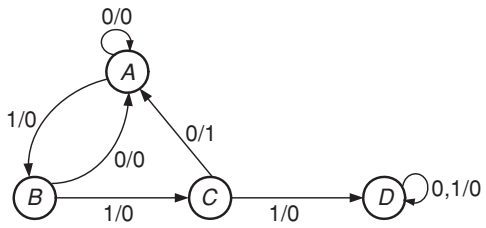


Fig. 10.1 State diagram for machine *M*.

where I, O, and S are finite nonempty sets of inputs, outputs, and states, respectively:

δ: $I \times S \rightarrow S$ is the state transition function;
λ is the output function such that
λ: $I \times S \rightarrow O$ for Mealy machines;
λ: $S \rightarrow O$ for Moore machines.

The Cartesian product $I \times S$ is the set containing all pairs of elements (I_i, S_j). The state transition function δ associates with each pair (I_i, S_j) an element S_k from S called the *next state*. In a Mealy machine the output function λ associates with each pair (I_i, S_j) an element O_k from O, while in a Moore machine a correspondence exists between the states and outputs.

Input–output transformations

Consider the machine M whose state diagram is given in Fig. 10.1. It is a four-state machine, with one input variable and one output variable, for which

$$S = \{A, B, C, D\}, \quad I = \{0, 1\}, \quad O = \{0, 1\}.$$

Suppose that the initial state of M is A and the input sequence is 110. Then the machine will proceed through states B and C and return to state A, while producing the output sequence 001. Thus, for an initial state A, the machine M *transforms* the input sequence 110 into 001. Similarly, for the same initial state, the input sequence 01100 is transformed into 00010. Since every computation involves some transformation of input-to-output sequences, a finite-state machine is capable of performing a variety of computations and solving a number of problems that can be expressed as a transformation of sequences.

An important function of a sequential machine is to determine whether a given input sequence is a member of some prespecified set of sequences. The machine accomplishes this function by accepting those sequences that are members of the set and rejecting those that are not. A machine, when started in its initial state, *accepts* an input sequence by producing an output value 1 as it receives the last symbol of that sequence. Thus machine M accepts the

sequences 110 and 0110 and rejects the sequence 01100, since its corresponding last output symbol is 0. The sequence detector of Fig. 9.15 can also be described as a machine that accepts those input sequences that are members of the set {all sequences whose last four symbols are 0101}.

The general problem of characterizing a machine's behavior by observing its input–output transformations is quite complex. Clearly, it is impractical to feed a machine with all possible input sequences in order to decide which it accepts. The problem increases in complexity if we wish to determine whether two arbitrary machines are related, in the sense that one machine accepts all the sequences accepted by the other. In this chapter, we shall present finite experiments to determine the characteristics, capabilities, and limitations of a machine and the relations between machines. These subjects are further developed in Chapters 13, 14, and 16.

Returning to the state diagram for M, we note that the application of input symbol 1 to M, when initially in state A, causes a transition to state B. We thus say that B is the 1-successor of A. In general, if an input sequence X takes a machine from state S_i to S_j then S_j is said to be the *X-successor* of S_i. For example, state D is the 111-successor of A. If M is known to be initially in either state B or C, the 10-successor will be either state A or D. We say that (AD) is the 10-successor of (BC) if A is the 10-successor of B and D that of C.

It is evident that no input sequence exists that can take M out of state D, and thus D is said to be a terminal state. Generally, a state is called *terminal* if either of the following is true: (i) the corresponding vertex in the state diagram is a *sink* vertex, i.e., no outgoing arcs that emanate from it terminate in other vertices; (ii) the corresponding vertex is a *source*, i.e., no arcs that emanate from other vertices terminate in it.

A source state is clearly not accessible from any other state and, similarly, no state is accessible from a sink state. These are extreme examples of situations that limit the state transitions in a sequential machine. In other cases, certain subsets of states may not be reachable from other subsets of states, even if the machine does not contain any terminal state. If, for every pair of states S_i, S_j of a machine M, there exists an input sequence that takes M from S_i to S_j then M is said to be *strongly connected*. Clearly, any nontrivial machine that has terminal states is not strongly connected.

10.2 Capabilities and limitations of finite-state machines

At this point, having established several behavioral properties and synthesis procedures for finite-state machines, we turn our attention to some basic questions regarding the capabilities of these machines. What can a machine do? Are there any limitations on the type of input–output transformations that can be performed by a machine? What restrictions are imposed on the capabilities of

the machine by the finiteness of the number of its states? Although a precise answer to these questions will be deferred to Chapter 16, we will point out the existence of problems not solvable by any finite-state machine and determine a characteristic of the transformations that are realizable by such machines.

Let the input to an n-state machine be an arbitrarily long sequence of 1's. In response, the machine will progress, starting from some initial state, through a succession of states, in accordance with its specified state transitions. Now, if we let the sequence be longer than n, the machine must eventually arrive at a state in which it has previously been. Consequently, from this point on, and because the input symbol remains the same, the machine must continue in a periodically repeating fashion. Clearly, for an n-state machine the period cannot exceed n and could be smaller. Moreover, the transient time until the output reaches its periodic pattern cannot exceed the number of states n. The preceding result can easily be generalized to any arbitrary input sequence consisting of a string of repeated symbols. In every such case, the output will become periodic after a transient time no longer than n.

This conclusion leads to many interesting results that exhibit the limitations of finite-state machines. For example, suppose that we want to design a machine which receives a long sequence of 1's and is to produce output symbol 1 when and only when the number of input symbols that it has received so far is equal to $k(k+1)/2$, for $k = 1, 2, 3, \ldots$ That is, the desired input–output transformation has the following form:

input	=	1	1	1	1	1	1	1	1	1	1	1	1	1	1	$\cdots$
output	=	1	0	1	0	0	1	0	0	0	1	0	0	0	0	1	$\cdots$

Clearly, since the output sequence does not eventually become periodic, no finite-state machine can produce such an infinite sequence.

In Section 9.1 we designed a serial adder capable of serially adding two binary numbers of arbitrary length. As another example demonstrating the limitations on the capabilities of finite-state machines, we shall show that the serial-multiplication problem is not solvable by a fixed finite-state machine, i.e., *no finite-state machine with a fixed number of states can multiply two arbitrarily large binary numbers.*

To prove the foregoing assertion, suppose that there does exist an n-state machine capable of serially multiplying any two binary numbers. Let us select 2^p as each of the two numbers to be multiplied, so that $2^p \times 2^p = 2^{2p}$, where $p > n$. The inputs are fed serially into the machine, least significant digits first: 2^p is represented by a 1 followed by p 0's, and 2^{2p} by a 1 followed by $2p$ 0's. The input symbols are fed into the machine during the first $p+1$ time units, i.e., between t_1 and t_{p+1}, as shown in the table below. During this period, the machine produces 0's. At t_{p+1} the input stops, while the machine must go on producing p additional 0's followed by a 1.

t_{2p+1}	t_{2p}	$\cdots$	t_{p+1}	t_p	$\cdots$	t_2	t_1	time
			1	0	$\cdots$	0	0	first number
			1	0	$\cdots$	0	0	second number
1	0	$\cdots$	0	0	$\cdots$	0	0	product

During the time period between t_{p+1} and t_{2p} the machine receives no input but, since $p > n$, it must have been at one of the states twice during that time. Following the same line of argument as that pursued earlier, we are led to the conclusion that its output must be periodic and the period is smaller than p. Therefore, the machine will never produce the required output symbol 1.

Note that, for any two finite numbers, we can find a machine that is capable of multiplying them. However, the preceding result demonstrates that, for every finite-state machine capable of performing serial multiplication, we can find finite numbers that it cannot multiply. The reason for this limitation stems from the limited "memory" available to the machine. While in performing addition it only had to store information regarding a single-digit carry, in the multiplication problem it must be able to store arbitrarily large partial products.

In a similar manner, we can show that no finite-state machine with a fixed number of states can perform, for arbitrarily large size blocks, the computation executed by the Turing machine of Section 9.5.

As mentioned earlier, a more general and precise study of the capabilities and limitations of finite-state machines is deferred to Chapter 16, where they will be defined in terms of regular expressions.

10.3 State equivalence and machine minimization

In constructing the state diagram (or table) for a finite-state machine, it often happens that the diagram contains redundant states, i.e., states whose functions can be accomplished by other states. We note that the number of memory elements required for the realization of a machine is directly related to the number of states. (Recall that, for an n-state machine, $k = \lceil \log_2 n \rceil$ state variables are needed for an assignment.) Consequently, the minimization of the number of states does reduce the complexity and cost of the realization in many cases. Moreover, the testing of sequential machines, which is studied in Chapter 13, is considerably simpler when the machine does not contain redundant states. It is, therefore, desirable to develop techniques for transforming a given machine into another machine that has no redundant states, such that both have the same terminal behavior.

Table 10.1 (*a*) Machine M_1 and (*b*) its state partitions

	NS, z	
PS	$x = 0$	$x = 1$
A	$E, 0$	$D, 1$
B	$F, 0$	$D, 0$
C	$E, 0$	$B, 1$
D	$F, 0$	$B, 0$
E	$C, 0$	$F, 1$
F	$B, 0$	$C, 0$

(*a*)

Symbol	Partition
P_0	(ABCDEF)
P_1	(ACE)(BDF)
P_2	(ACE)(BD)(F)
P_3	(AC)(E)(BD)(F)
P_4	(AC)(E)(BD)(F)

(*b*)

The *k*-equivalence of states

Two states, S_i and S_j, of a machine M are *distinguishable* if and only if there exists at least one finite input sequence that, when applied to M, causes different output sequences depending on whether S_i or S_j is the initial state. The sequence that distinguishes these states is called a *distinguishing sequence for the pair* (S_i, S_j). If there is any uncertainty as to whether the state of M is S_i or S_j then an application of the corresponding distinguishing sequence yields an output sequence that is sufficient to determine the unknown state uniquely. If there exists a distinguishing sequence of length k for the pair (S_i, S_j), the states S_i, S_j are said to be *k-distinguishable*.

As an example, consider pair (A, B) of the machine M_1 whose state table is shown in Table 10.1*a*. The pair (A, B) is 1-distinguishable, since the input symbol 1 applied to M_1 when initially in state A yields the output symbol 1 and when initially in state B yields the output symbol 0. However, the pair (A, E) is 3-distinguishable since there is no input sequence of length less than 3 that distinguishes A from E. Furthermore, the only sequence of length 3 that is a distinguishing sequence for the pair (A, E) is $X = 111$, and the output sequences corresponding to the initial states A and E are 100 and 101, respectively. Note that 1101 is also a sequence that distinguishes A from E, although it is not the shortest such sequence. An all-zero sequence will produce identical output sequences independently of whether the initial state is A or E.

The concept of k-distinguishability leads directly to the definition of k-equivalence and equivalence. States that are not k-distinguishable are said to be *k-equivalent*. For example, states A and E of M_1 are 2-equivalent. States that are k-equivalent are also r-equivalent, for all $r < k$. States that are k-equivalent for all k are said to be *equivalent*. Thus, we arrive at the following definition.

Definition 10.2 The states S_i and S_j of machine M are said to be *equivalent* if and only if, *for every possible input sequence*, the same output sequence is produced regardless of whether S_i or S_j is the initial state.

Thus, S_i and S_j are equivalent (indicated by $S_i = S_j$) if there is no input sequence that distinguishes them. It will be subsequently shown (see Theorem 10.2) that states which are k-equivalent for all $k \leq n - 1$ are equivalent. Clearly, if $S_i = S_j$ and $S_j = S_k$ then $S_i = S_k$. It therefore follows (see Section 2.2) that state equivalence is an equivalence relation. In consequence of this characteristic, the set of states of the machine can be partitioned into disjoint subsets, known as *equivalence classes*, such that two states are in the same equivalence class if and only if they are equivalent and are in different classes if and only if they are distinguishable. Definition 10.2 can be generalized to the case where S_i is a possible initial state in machine M_1 while S_j is an initial state in machine M_2, where both M_1 and M_2 have the same input alphabet.

The procedure for determining the sets of equivalent states in a machine, i.e., the equivalence classes, ensues from the following property. *If S_i and S_j are equivalent states then their corresponding X-successors, for all X, are also equivalent*. This follows since otherwise it would be trivial to construct a distinguishing sequence for (S_i, S_j) by first applying an input sequence that transfers the machine to the distinguishable successors of S_i and S_j.

The minimization procedure

The object of this section is to describe a procedure for determining the sets of equivalent states of a specified machine M. The result sought is a partition on the states of M such that two states are in the same block if and only if they are equivalent.

The first step is to partition the states of M into subsets such that all states in the same subset are 1-equivalent. This is accomplished by placing states having identical output symbols under all possible input symbols in the same subset. Clearly, two states that are in different subsets are 1-distinguishable. As an example, consider the partitions of the states of machine M_1 given in Table 10.1*b*. The first partition P_0 corresponds to 0-distinguishability and defines our initial "ignorance," regarding the response of the various states, prior to the application of any input symbol. The partition P_1 is obtained simply by inspecting the table and placing in the same block states having the same output symbols for all input symbols. Thus A, C, and E are in the same block since their output symbols, for input symbols 0 and 1, are 0 and 1, respectively. A similar argument places B, D, and F in the other block. Clearly, P_1 establishes the sets of states that are 1-equivalent.

The next step is to obtain the partition P_2 whose blocks consist of the sets of states which are 2-equivalent, that is, equivalent under any input sequence of length 2. This is accomplished by observing that two states are 2-equivalent if and only if they are 1-equivalent and their I_i-successors, for all possible I_i, are also 1-equivalent. Consequently, two states are placed in the same block of P_2 if and only if they are in the same block of P_1 and, for each possible I_i, their I_i-successors are also contained in a block of P_1. This step is carried out

by splitting the blocks of P_1 whenever their successors are not contained in a common block of P_1. The 0- and 1-successors of (ACE) are (CE) and (BDF), respectively, and, since both are contained in common blocks of P_1, the states in (ACE) are 2-equivalent and therefore (ACE) constitutes a block in P_2. The 1-successor of (BDF) is (DBC) but, since (DB) and (C) are not contained in a single block of P_1, block (BDF) must be split into (BD) and (F) in such a way that the successors of the blocks in the refined[1] partition are 1-equivalent. In a similar manner P_3 is obtained by splitting block (ACE) of P_2 into (AC) and (E), since the 1-successors of A, C, and E are D, B, and F, which are not 2-equivalent.

In general, the partition P_{k+1} is obtained from P_k by placing in the same block of P_{k+1} those states that are in the same block of P_k *and* whose I_i-successors for every possible I_i are also in a common block of P_k. This process places the states that are $(k+1)$-equivalent in the same block and states that are $(k+1)$-distinguishable in different blocks. Note that no state can belong to more than one block since this would make it distinguishable with respect to itself.

If, for some k, $P_{k+1} = P_k$ then the process terminates and P_k defines the sets of equivalent states of the machine; that is, all states contained in the same block of P_k are equivalent while states belonging to different blocks are distinguishable. The partition P_k is thus called the *equivalence partition*, and the foregoing procedure is referred to as the *Moore reduction procedure*. For the machine M_1, the equivalence partition is P_3 and therefore states A and C are equivalent and so are B and D. Before proceeding with the minimization procedure, we shall prove two theorems to establish its validity and determine its length.

Theorem 10.1 *The equivalence partition is unique.*

Proof Suppose that there exist two equivalence partitions P_a and P_b and that $P_a \neq P_b$. Then there exist two states S_i and S_j that are in the same block of one partition and are not in the same block of the other. Since S_i and S_j are in different blocks of (say) P_b, there exists at least one input sequence that distinguishes S_i from S_j and, therefore, they cannot be in the same block of P_a. ◇

Theorem 10.2 *If two states S_i and S_j of machine M are distinguishable then they are distinguishable by a sequence of length $n-1$ or less, where n is the number of states in M.*

Proof The partition P_1 contains at least two blocks; otherwise M would be reducible to a combinational circuit that has only a single state. At each step, the partition P_{k+1} is smaller than or equal to P_k. (Recall that a partition $P_i \leq P_j$ if every block of P_i is contained in a block of P_j; e.g., P_2 of M_1 is smaller

[1] A partition P is said to be a *refinement* of a partition Q if P is smaller than Q.

Table 10.2 Machine M_1^*

	NS, z	
PS	$x = 0$	$x = 1$
α	$\beta, 0$	$\gamma, 1$
β	$\alpha, 0$	$\delta, 1$
γ	$\delta, 0$	$\gamma, 0$
δ	$\gamma, 0$	$\alpha, 0$

than P_1.) If P_{k+1} is smaller than P_k then it contains at least one more block than P_k. However, since the number of blocks is limited to n, at most $n - 1$ partitions can be generated in the reduction procedure and, thus, if S_i and S_j are distinguishable then they are distinguishable by a sequence of length $n - 1$ or smaller. ◇

It can be shown (see Problem 10.15) that the above is indeed the least upper bound.

Machine equivalence

Before proceeding with the determination of the minimal machine that is equivalent to M_1, we shall define precisely what we mean by equivalent and minimal machines.

Definition 10.3 Two machines M_1 and M_2 are said to be *equivalent* if and only if for every state in M_1 there is a corresponding equivalent state in M_2 and vice versa.

The equivalence partition has been shown to be unique. Thus, the number of blocks in the equivalence partition of a machine M defines the *minimum* number of states that any machine equivalent to M must have. The machine that contains no equivalent states and is equivalent to M is called the *minimal*, or *reduced*, form of M.

If we denote the blocks of the equivalence partition P_3 of M_1 by α, β, γ, and δ, corresponding respectively to (AC), (E), (BD), and (F), we obtain the machine M_1^* (Table 10.2). In constructing M_1^*, we specify the 1-successor of α to be γ, since the 1-successor of (AC) is (BD), and so on. In this manner, M_1^* is specified to duplicate the state transitions and response of M_1 and, therefore, is equivalent to it. In addition, since it has been generated by the equivalence partition of M_1, it is its minimal form.

Example We shall illustrate the reduction procedure further by applying it to a machine M_2 (Table 10.3) and finding its minimal form. The blocks

of the equivalence partition P_4 are denoted $\alpha, \beta, \ldots, \epsilon$, and the reduced machine M_2^* (Table 10.4) results.

Table 10.3 (*a*) Machine M_2 and (*b*) its state partition

	NS, z	
PS	$x = 0$	$x = 1$
A	$E, 0$	$C, 0$
B	$C, 0$	$A, 0$
C	$B, 0$	$G, 0$
D	$G, 0$	$A, 0$
E	$F, 1$	$B, 0$
F	$E, 0$	$D, 0$
G	$D, 0$	$G, 0$

(*a*)

Symbol	Partition
P_0	(ABCDEFG)
P_1	(ABCDFG)(E)
P_2	(AF)(BCDG)(E)
P_3	(AF)(BD)(CG)(E)
P_4	(A)(F)(BD)(CG)(E)
P_5	(A)(F)(BD)(CG)(E)

(*b*)

Table 10.4 Machine M_2^*

		NS, z	
PS		$x = 0$	$x = 1$
(A)	α	$\epsilon, 0$	$\delta, 0$
(F)	β	$\epsilon, 0$	$\gamma, 0$
(BD)	γ	$\delta, 0$	$\alpha, 0$
(CG)	δ	$\gamma, 0$	$\delta, 0$
(E)	ϵ	$\beta, 1$	$\gamma, 0$

The selection of labels $\alpha, \beta, \ldots$ assigned to the blocks of P_4 is obviously arbitrary. A different assignment of labels would have described a machine with the same behavioral properties. In general, if one machine can be obtained from the other by relabeling its states then they are said to be *isomorphic* to each other. The foregoing results lead to the following basic conclusion:

- To every machine M there corresponds a minimal machine M^* that is equivalent to M and is unique up to isomorphism.

The detection of isomorphism is not always easy and is best accomplished by using a canonical representation for a machine. Such a representation is obtained by selecting a state (preferably the starting state if specified) and labeling it A. The next labels are selected in such a way that when successive rows of the table, starting in A and going down through B, C, etc., are read from left to right, the first occurrence of each new label will be in alphabetical order. Whenever a machine is given in this canonical representation, it is said to be in *standard form*. Clearly, when the starting state of a reduced machine is specified, its standard form is unique.

Table 10.5 Standard form for M_2^*

		NS, z	
PS		$x = 0$	$x = 1$
α	A	$B, 0$	$C, 0$
ϵ	B	$D, 1$	$E, 0$
δ	C	$E, 0$	$C, 0$
β	D	$B, 0$	$E, 0$
γ	E	$C, 0$	$A, 0$

The transformation of a machine into its standard form will be illustrated by means of M_2^*. Denoting α by A implies that its 0-successor ϵ must be denoted B, because it is the first occurrence of a new label. Similarly, its 1-successor δ must be denoted C. Row B (i.e., ϵ) must be relabeled next; its first entry is β and, since it is a new label, it is denoted D. Similarly, γ is denoted E, and the standard form of Table 10.5 results.

When the starting states are not specified the detection of isomorphism is, in general, not as simple. If the number of states is not too large, however, isomorphism can be detected by inspecting the state diagrams of the machines. The necessary and sufficient condition for two machines to be isomorphic to each other is that their state diagrams are identical except for the labeling of their vertices.

10.4 Simplification of incompletely specified machines

In practice, it often happens that various combinations of states and input symbols are not possible. For example, the machine of Table 9.15, when in state A, will never receive input symbol 0 and, consequently, the corresponding transition and its associated output symbol may be left unspecified. In other situations the state transitions are completely defined but, for some combinations of states and input symbols, the output values may not be critical and thus are left unspecified. Such machines are said to be *incompletely specified*; the determination of their properties and methods for simplifying them are the subject of this section.

Whenever a state transition is unspecified the future behavior of the machine may become unpredictable. In order to avoid such a situation, we shall assume that the input sequences applied to the machine, when in any of its possible starting states, are such that no unspecified next state is encountered except possibly at the final step. Such an input sequence is said to be *applicable* to the starting state S_i of M. Note that the output symbols encountered need not

Table 10.6 Machine M_3 with unspecified transitions

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 1$	—
B	–, 0	$C, 0$
C	$A, 1$	$B, 0$

Table 10.7 An equivalent description where all transitions are specified

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 1$	T, –
B	$T, 0$	$C, 0$
C	$A, 1$	$B, 0$
T	T, –	T, –

all be specified for a sequence to be applicable to S_i. The next states, however, must be specified except possibly for the last symbol of the sequence.

Actually, the specified behavior of a machine with partially specified transitions can be described by another machine whose state transitions are completely specified. This transformation is accomplished by adding a terminal state T whose output symbols are unspecified and replacing all the dashes in the next-state entries by T. As an illustration, consider the machine M_3 shown in Table 10.6. The specified behavior of M_3 can be described by Table 10.7, in which all state transitions are specified and only the output symbols are partially defined.

Compatible states

In Section 10.3 we defined state and machine equivalence. We shall find it useful to generalize these concepts as follows.

Definition 10.4 State S_i of M_1 is said to *cover*, or *contain*, state S_j of M_2 if and only if every input sequence applicable to S_j is also applicable to S_i and its application to both M_1 and M_2 when they are initially in S_i and S_j, respectively, results in identical output sequences whenever the output symbols of M_2 are specified.

This covering concept can be extended to machines as follows. *Machine M_1 is said to cover machine M_2 if and only if, for every state S_j in M_2, there is a corresponding state S_i in M_1 such that S_i covers S_j*. Clearly the machine specified by Table 10.6 is covered by that of Table 10.7. If state S_i of machine M covers another state S_j of the same machine then only S_i must be retained; S_j may be deleted.

Definition 10.5 Two states S_i and S_j of a machine M are *compatible* if and only if, for every input sequence applicable to both S_i and S_j, the same output sequence will be produced *whenever both output symbols are specified* and regardless of whether S_i or S_j is the initial state.

Hence S_i and S_j are compatible if and only if their output symbols are not conflicting (i.e., identical when specified) and their I_i-successors, for every I_i for which both are specified, are either the same or also compatible. In general, three or more states, $S_i, S_j, S_k, \ldots$, are compatible if and only if, for every applicable input sequence, no two conflicting output sequences will be produced, without regard as to which of the above states is the initial state. Thus, a set of states $(S_i, S_j, S_k, \ldots)$ is called a *compatible* if all its members are compatible.

A compatible C_i is said to be *larger* than, or to *cover*, another compatible C_j if and only if every state contained in C_j is also contained in C_i. A compatible is *maximal* if it is not covered by any other compatible. (Note that a single state that is not compatible with any other state is a maximal compatible.) Thus, if we find the set of all the maximal compatibles, this in effect is equivalent to finding all compatibles since every subset of a compatible is also a compatible.

Generalizing slightly, we find that, in the case of incompletely specified machines, the analog to the equivalence relation studied earlier is the compatibility relation. The similarities and differences between these two relations will be pointed out subsequently.

The nonuniqueness of the reduced and minimal machines

Before developing the simplification procedure for incompletely specified machines, we shall illustrate some difficulties encountered in applying the minimization procedure of Section 10.3 to the machine M_4 shown in Table 10.8.

The dashes in row A, column 1, and in row B, column 0, mean that the output symbols associated with these transitions will be ignored and thus may be specified according to our convenience. If we replace both dashes by 1's, we find that states A and B become equivalent since their output symbols and corresponding successors are identical. Consequently, we may combine these states by redirecting to A all the transitions presently leading to B. The resulting simplified machine, shown in Table 10.9, is in reduced form and thus cannot be further simplified. If, however, we choose to specify the dashes as 0's then it is easy to verify that states A and E are equivalent, and in addition states B, C, and D become equivalent. Thus, we may relabel blocks (AE) and (BCD) by α and β, respectively, and the minimal machine of Table 10.10 results.

From the foregoing example, the following observations can be made. States A and B of M_4 are compatible and, if C and D are also compatible, so are A and E. However, states B and E are 1-distinguishable and, therefore, incompatible. Consequently, since it is not transitive the compatibility relation is not an equivalence relation. It thus follows that *a set of states is a compatible if and only if every pair of states in that set is compatible*. For example, states B, C,

Table 10.8 Machine M_4

	NS, z	
PS	$x = 0$	$x = 1$
A	$C, 1$	$E, -$
B	$C, -$	$E, 1$
C	$B, 0$	$A, 1$
D	$D, 0$	$E, 1$
E	$D, 1$	$A, 0$

Table 10.9 A simplified reduced machine, M_4^*

	NS, z	
PS	$x = 0$	$x = 1$
A	$C, 1$	$E, 1$
C	$A, 0$	$A, 1$
D	$D, 0$	$E, 1$
E	$D, 1$	$A, 0$

Table 10.10 A minimal machine, $M_4^\#$

		NS, z	
PS		$x = 0$	$x = 1$
(AE)	α	$\beta, 1$	$\alpha, 0$
(BCD)	β	$\beta, 0$	$\alpha, 1$

and D of M_4 form the compatible (BCD), since (BC), (BD), and (CD) are compatibles.

The machines M_4^* and $M_4^\#$ both cover M_4, and their numbers of states are each smaller than the number of states of M_4. Both are in reduced form; i.e., they contain no redundant states. This situation, in which two different reduced machines cover a third one, is evidently in contrast with Theorem 10.1. This poses a serious difficulty in applying the previously derived minimization procedure, since we can no longer be content with finding a reduced machine covering the original one; our aim must be to find a reduced machine that not only covers the original machine but also has a minimal number of states.

A further and crucial difference between completely and incompletely specified machines is demonstrated by means of machine M_5 (Table 10.11). Because of the output entries, the only candidates for equivalence are the states A and B or B and C. Also, because of the next-state entries, A is equivalent to B only if B is equivalent to C. However, for A and B to be equivalent the dash must be replaced by a 0 while for B and C to be equivalent the dash must be replaced by a 1. Evidently, there is no way of specifying the unspecified entry so as to achieve any state equivalence. However, a hasty conclusion that M_5 is in reduced form would be false, as is shown subsequently.

The augmented machine of Table 10.12 is obtained by a process known as *state splitting*. This process involves the replacement of a state S_i by two or more states $S_i', S_i'', \ldots$ such that each new state covers S_i. To ensure that the

Table 10.11 Machine M_5

	NS, z	
PS	$x = 0$	$x = 1$
A	$A, 0$	$C, 0$
B	$B, 0$	$B, -$
C	$B, 0$	$A, 1$

Table 10.12 Augmented machine

	NS, z	
PS	$x = 0$	$x = 1$
A	$A, 0$	$C, 0$
B'	$B', 0$	$B'', -$
B''	$B^+, 0$	$B', -$
C	$B^+, 0$	$A, 1$

Table 10.13 Two minimal machines corresponding to M_5

		NS, z	
PS		$x = 0$	$x = 1$
(AB')	α	$\alpha, 0$	$\beta, 0$
$(B''C)$	β	$\alpha, 0$	$\alpha, 1$

(a) Setting $B^+ = B'$

		NS, z	
PS		$x = 0$	$x = 1$
(AB')	α	$\alpha, 0$	$\beta, 0$
$(B''C)$	β	$\beta, 0$	$\alpha, 1$

(b) Setting $B^+ = B''$

augmented machine covers the original one, it is necessary to modify the next-state entries in such a way that each transition to S_i is replaced by a transition to either S_i' or S_i'', etc. In our case, state B has been split into B' and B'' and the next-state entries modified as shown in Table 10.12, where the symbol B^+ means that the transition may be either B' or B''. Clearly, the augmented machine covers M_5 and is reducible to it by letting $B' = B'' = B$.

In general, since B' and B'' both cover B, we may specify the next-state entries B arbitrarily as B' or B''. If, however, we select the specification shown in Table 10.12 then a simplification of M_5 becomes possible. States A and B' are compatible if their 1-successors C and B'' are compatible. Similarly, states B'' and C are compatible if their 1-successors B' and A are compatible. Thus, if we designate the compatibles (AB') and $(B''C)$ by α and β, respectively, we obtain the minimal machines of Table 10.13. The result is Table 10.13a or 10.13b, depending on whether B^+ is specified as B' or B''.

The foregoing example demonstrates the nonuniqueness of the minimal machine in the case of incompletely specified machines. The minimal machines of Table 10.13 were obtained by allowing state B to be split in such a way that it can be made equivalent to both A and C (by specifying the unspecified output symbol differently). This points out the main difference between completely and incompletely specified machines. *While the equivalence partition consists of disjoint blocks, the subsets of compatibles may be overlapping.*

Table 10.14 Machine M_6

	NS, z			
PS	I_1	I_2	I_3	I_4
A	—	$C, 1$	$E, 1$	$B, 1$
B	$E, 0$	—	—	—
C	$F, 0$	$F, 1$	—	—
D	—	—	$B, 1$	—
E	—	$F, 0$	$A, 0$	$D, 1$
F	$C, 0$	—	$B, 0$	$C, 1$

The merger graph

In reducing the machine M_4, we actually specified the don't-care entries and thus transformed the incompletely specified machine into a completely specified one. Such a specification may not be optimal and then would drastically reduce our freedom in simplifying the machine. It is, therefore, desirable first to generate the entire set of compatibles and then to select an appropriate subset, which will form the basis for a state reduction leading to a minimal machine.

Since a set of states is compatible if and only if every pair of states in that set is compatible, it is sufficient to consider only pairs of states and to use them to generate the entire set. We shall refer to a compatible pair of states as a *compatible pair*. Let the I_k-successors of S_i and S_j be S_p and S_q, respectively; then $(S_p S_q)$ is said to be implied by $(S_i S_j)$. For example, the compatible (CF) of machine M_6 (Table 10.14) is implied by (AC), and so on. Thus, if $(S_i S_j)$ is a compatible pair then $(S_p S_q)$ is referred to as its *implied pair*. In general, a set of states P is *implied* by a set of states Q if, for some input symbol I_k, P is the set of all I_k-successors of the states in Q. The merger graph, presented below, serves as a major tool in the determination of the set of all compatibles.

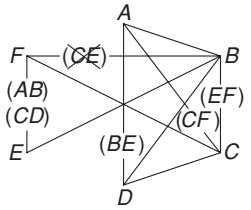


Fig. 10.2 Merger graph for M_6.

The *merger graph* of an n-state machine M is an undirected graph defined as follows.

1. It consists of n vertices, each of which corresponds to a state of M.
2. For each pair of states $(S_i S_j)$ in M, whose next-state and output entries are not conflicting, an undirected arc is drawn between the vertices S_i and S_j.
3. If, for a pair of states $(S_i S_j)$, the corresponding output symbols under all input symbols are not conflicting but the successors are not the same, an interrupted arc is drawn between S_i and S_j and the implied pairs are entered in the space.

Consider the machine M_6 (Table 10.14) and its merger graph, shown in Fig. 10.2. Since the next-state and output entries of states A and B are not conflicting, an arc is drawn between vertices A and B. States A and C, however, have nonconflicting output symbols but their successors under the input symbol I_2 are C and F. Therefore, (AC) is a compatible only if (CF) is; consequently,

an interrupted arc is drawn between the vertices A and C and (CF) is entered in the space. Similarly, (AD) is a compatible only if (BE) is, and thus (BE) is entered in the space of the interrupted arc drawn between A and D. No arc is drawn between A and E since these states are incompatible, their output symbols under I_2 and I_3 being conflicting. In a similar manner, every possible pair of states is checked, and the entire merger graph obtained.

A merger graph displays all possible pairs of states and their implied pairs, and since a pair of states is compatible only if its implied pair is, one must now check to determine whether the implied pairs are indeed compatibles. A pair (S_pS_q) is incompatible if no arc is drawn between vertices S_p and S_q. In such a case, if (S_pS_q) is written in the space of an interrupted arc, entry (S_pS_q) is crossed off and the corresponding arc ignored. For example, in Fig. 10.2 the condition for (BF) to be compatible is that (CE) be compatible but, since there is no arc drawn between C and E, (CE) is incompatible and the arc between B and F is ignored. Thus, states B and F are incompatible. Next it is necessary to check whether the incompatibility of (BF) invalidates any other implied pair, that is, whether (BF) is written in the space of another interrupted arc, and so on. The interrupted arcs that remain in the graph, after all the implied pairs have been verified to be compatible, are regarded as solid ones.

For the machine M_6, the merger graph reveals the existence of nine compatible pairs:

$$(AB),\quad (AC),\quad (AD),\quad (BC),\quad (BD),\quad (BE),\quad (CD),\quad (CF),\quad (EF)$$

Moreover, since (AB), (AC), and (BC) are compatibles then (ABC) is also a compatible, and so on. In this manner, the entire set of compatibles of M_6 can be generated from its compatible pairs.

In order to find a minimal set of compatibles, which covers the original machine and can be used as a basis for the construction of a minimal machine, it is often useful to find the set of maximal compatibles. Recall that a compatible is maximal if it is not contained in any other compatible. In terms of the merger graph, we are looking for complete polygons that are not contained within any higher-order complete polygons. (A *complete polygon* is one in which all possible $(n-3)n/2$ diagonals exist, where n is the number of sides in the polygon.) Since the states covered by a complete polygon are all pairwise compatible, they constitute a compatible; and, if the polygon is not contained in any higher-order complete polygon, they constitute a maximal compatible.

In Fig. 10.2 the set of highest-order polygons are the tetragon $(ABCD)$ and the arcs (CF), (BE), and (EF). Generally, after a complete polygon of order n has been found, all polygons of order $n-1$ contained in it can be ignored. Consequently, the triangles (ABC), (ACD), etc., are not considered. Thus, the following set of maximal compatibles for machine M_6 results:

$$\{(ABCD), (BE), (CF), (EF)\}$$

The closed sets of compatibles

Consider the set of compatibles $\{(ABCD), (EF)\}$ of machine M_6. Since this is the minimal number of compatibles covering all the states of M_6, it defines a *lower bound* on the number of states in the minimal machine that covers M_6. However, if we select the maximal compatible $(ABCD)$ to be a state in the reduced machine, its I_2- and I_3-successors, (CF) and (BE), respectively, must also be selected. Since none of these compatible pairs is contained in the above set the lower bound cannot be achieved, and the set of maximal compatibles $\{(ABCD), (EF)\}$ cannot be used to define the states of a minimal machine that covers M_6.

Definition 10.6 A set of compatibles (for a machine M) is said to be *closed* if, for every compatible contained in the set, all its implied compatibles are also contained in the set. A closed set of compatibles that contains all the states of M is called a *closed covering*.

Example For M_6, the set $\{(AD), (BE), (CD)\}$ is closed. The set $\{(AB), (CD), (EF)\}$ is a closed covering.

For incompletely specified machines, the closed covering serves the same function as that served by the equivalence partition for completely specified machines. It specifies the states that are compatible and may be covered by a single state of a reduced machine. However, as demonstrated by the preceding examples, the closed covering is not unique and so our task is to select the one which has a minimum number of compatibles and thus defines a minimal-state machine that covers the original one.

The set containing all the maximal compatibles is, clearly, a closed covering since it covers all the states of the machine and every implied compatible is contained in the set. Consequently, the set of maximal compatibles places an *upper bound* on the number of states in the machine that cover the original state. For machine M_6, this upper bound is four. It must be noted at this point that the concept of an upper bound is meaningless when the number of maximal compatibles is larger than the number of states in the original machine.

In the preceding discussion, we showed that the bounds on the number of states in the minimal machine can be derived from the set of all the maximal compatibles. For machine M_6, these bounds were found to be two and four. However, since the lower bound cannot be achieved it becomes necessary to determine whether a closed covering containing three compatibles can be found. These compatibles need not necessarily be maximal; in fact, the maximal compatible $(ABCD)$ cannot be included in that set since it implies the entire set of maximal compatibles.

An inspection of the merger graph of Fig. 10.2 reveals that states A and B can be covered by the compatible pair (AB) and, similarly, states C and D

Table 10.15 A minimal machine covering M_6

		NS, z			
PS		I_1	I_2	I_3	I_4
(AB)	α	$\gamma, 0$	$\beta, 1$	$\gamma, 1$	$\alpha, 1$
(CD)	β	$\gamma, 0$	$\gamma, 1$	$\alpha, 1$	—
(EF)	γ	$\beta, 0$	$\gamma, 0$	$\alpha, 0$	$\beta, 1$

Table 10.16 Machine M_7

	NS, z			
PS	I_1	I_2	I_3	I_4
A	—	—	$E, 1$	—
B	$C, 0$	$A, 1$	$B, 0$	—
C	$C, 0$	$D, 1$	—	$A, 0$
D	—	$E, 1$	$B, -$	—
E	$B, 0$	—	$C, -$	$B, 0$

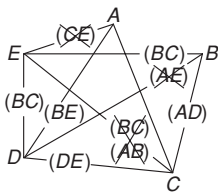


Fig. 10.3 Merger graph for M_7.

can be covered by (CD); no pairs are implied by these compatibles, which thus form a closed set. In order to obtain the desired covering, all we need is a single compatible that covers states E and F. Fortunately, the pair (EF) is compatible and implies the pairs (AB) and (CD), which are contained in the above set. Consequently, the set $\{(AB), (CD), (EF)\}$ is a closed covering containing three compatibles, and it thus yields a minimal three-state machine that covers M_6. This machine is shown in Table 10.15. In a similar manner, we can show that the set $\{(AD), (BE), (CF)\}$ is also a closed covering that corresponds to a minimal machine containing M_6.

The preceding closed coverings have been obtained by inspecting the merger graph and employing a "trial-and-error" procedure. In the following section, we shall discuss in detail a more systematic procedure for obtaining minimal closed coverings.

The compatibility graph

Consider the machine M_7 and its merger graph, shown in Table 10.16 and Fig. 10.3, respectively. The merger graph is constructed in the usual manner; since states A and B are incompatible, the arc between C and E is crossed off and, as a result, (AE) and (BD) are also found to be incompatible. The set of maximal compatibles derived from the merger graph contains four members and is given by

$$\{(ACD), (BC), (BE), (DE)\}.$$

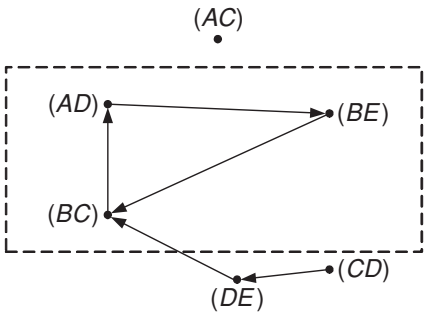


Fig. 10.4 Compatibility graph for M_7.

The *compatibility graph* is a directed graph whose vertices correspond to all compatible pairs and for which an arc leads from vertex $(S_i S_j)$ to vertex $(S_p S_q)$ if and only if $(S_i S_j)$ implies $(S_p S_q)$. It is a tool that aids our search for a minimal closed covering.

The compatible pairs and their implied pairs are usually obtained from the merger graph and, since a set of states is a compatible if and only if every pair of states in that set is compatible, then for a given machine the set of compatible pairs uniquely defines the entire set of compatibles.[2] In the compatibility graph of machine M_7 (Fig. 10.4), an arc leads from vertex (AD) to vertex (BE) because (AD) implies (BE). No arcs emanate from (AC) since no other compatible is implied by it.

A subgraph of a compatibility graph is said to be closed if, for every vertex in the subgraph, all outgoing arcs and their terminating vertices also belong to the subgraph. In addition, if every state of the machine is covered by at least one vertex of the subgraph then the subgraph forms a closed covering for that machine.

Example The compatibility graph of Fig. 10.4 contains seven closed subgraphs (including (AC) alone and the graph itself), six of which form closed coverings for M_7; among them, we find the subgraphs corresponding to the following coverings:

$$\{(BC), (AD), (BE)\}$$
$$\{(AC), (BC), (AD), (BE)\}$$
$$\{(DE), (BC), (AD), (BE)\}$$

The compatibility graph itself forms a closed covering. However, it is often desirable to look for a closed subgraph that yields a simpler machine. If a closed subgraph containing the compatible pairs $(S_i S_j)$, $(S_j S_k)$, and $(S_i S_k)$ has been found, the compatible $(S_i S_j S_k)$ can be formed, and so on. Although the number of states in the minimal machine is not necessarily proportional to the number

[2] In order to take into account states that are incompatible with all other states, the definition of the set of compatible pairs must be generalized to include the pairs corresponding to self-compatibility, i.e., (AA), (BB), etc.

Table 10.17 A minimal machine that covers M_7

		NS, z			
PS		I_1	I_2	I_3	I_4
(AD)	α	–	$\gamma, 1$	$\gamma, 1$	–
(BC)	β	$\beta, 0$	$\alpha, 1$	$\beta/\gamma, 0$	$\alpha, 0$
(BE)	γ	$\beta, 0$	$\alpha, 1$	$\beta, 0$	$\beta/\gamma, 0$

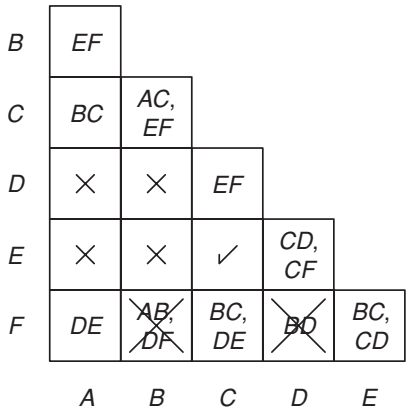


Fig. 10.5 Merger table for the machine M_8.

of vertices in the closed graph, the inclusion of many redundant vertices in it does tend to increase the size of the machine. A trial-and-error technique can be employed for this step. The compatibility graph thus serves to display the various possible reduced machines that correspond to the different closed coverings.

In the compatibility graph of the machine M_7, state B is covered by the vertices (BE) and (BC) and, since at least one of them must be included in any closed covering, the entire triangle $\{(BC), (AD), (BE)\}$ must also be included. This triangle, being a closed graph that covers every state of M_7, implies that the corresponding set of compatibles yields the desired minimal machine. Its state table is shown in Table 10.17, where the entry β/γ means that the next state may be either β or γ.

The merger table

When dealing with machines with a large number of states, it may be more convenient to record the compatible pairs and their implications in a merger table of the form illustrated in Fig. 10.5, instead of using a merger graph. Each cell of the table corresponds to the compatible pair defined by the intersection of the row and column headings. The incompatibility of two states is recorded by placing an $\times$ in the corresponding cell, while their compatibility is recorded

Table 10.18 Machine M_8

	NS, z	
PS	I_1	I_2
A	$E, 0$	$B, 0$
B	$F, 0$	$A, 0$
C	$E, -$	$C, 0$
D	$F, 1$	$D, 0$
E	$C, 1$	$C, 0$
F	$D, -$	$B, 0$

by a check mark ($\checkmark$). The entries in the cell S_i, S_j are the pairs implied by $(S_i S_j)$.

As an example, let us consider the machine M_8, whose state table is given in Table 10.18. Its merger table is shown in Fig. 10.5. An $\times$ is inserted in cell (AD) since states A and D have conflicting output symbols; a check mark is inserted in cell (CE) because state E contains state C. In a similar way the entire table is completed and the implied compatibles entered in the appropriate cells. Now it becomes necessary to check whether these entries indeed correspond to compatible pairs. Starting from the rightmost cell, we find no contradiction until we arrive at the entry (BD) in cell (DF). Since there is an $\times$ in cell (BD), the pair (DF) is incompatible and is, therefore, "crossed off." As a consequence of the incompatibility of (DF), the pair (BF) is also incompatible and the corresponding cell is crossed off.

Once the merger table has been completed, we continue to construct the corresponding compatibility graph and to find a closed subgraph, in order to obtain the smallest closed set of compatibles. Before continuing in the above-outlined direction, we shall pause and describe a procedure for finding the set of all maximal compatibles. This procedure is the tabular counterpart to that of finding complete polygons in the merger graph. It is executed in the following manner.

1. Start in the rightmost column of the merger table for the machine and proceed left until a column containing a compatible pair is encountered. List all the compatible pairs in that column. In our example, this step yields the pair (EF).
2. Proceed left to the next column containing at least one compatible pair. If the state to which this column corresponds is compatible with all members of some previously determined compatible, add this state to that compatible to form a larger compatible. If the state is not compatible with all members of a previously determined compatible but is compatible with some members of such a compatible, form a new compatible that includes those members and the state in question. Next, list all compatible pairs that are not included in any previously derived compatible.
3. Repeat step 2 until all columns have been considered. The final set of compatibles constitutes the set of maximal compatibles.

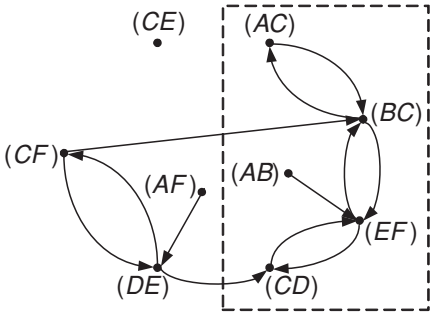


Fig. 10.6 Compatibility graph for M_8.

Applying this procedure to the merger table for machine M_8 yields the following sequence of compatibility classes:

column E, (EF);
column D, $(EF), (DE)$;
column C, $(CEF), (CDE)$;
column B, $(CEF), (CDE), (BC)$;
column A, $(CEF), (CDE), (ABC), (ACF)$.

From column C, it is evident that state C is compatible with states D, E, and F and consequently the compatibles generated previously are enlarged to include state C. Column B, however, consists of a single compatible pair, which is added to the previously generated list. From column A and rows B and C we obtain the compatible (ABC), while rows C and F, together with previously available compatibility relations, yield the compatible (ACF). The final list is the set of maximal compatibles of M_8.

The set of maximal compatibles clearly indicates that M_8 can be covered by a four-state machine and cannot be covered by any two-state machine. To determine whether a three-state machine that covers M_8 exists, we construct the compatibility graph shown in Fig. 10.6. It must be emphasized at this point that in many simple cases a shortcut can be taken, and the compatibility graph can be constructed directly from the state table, without the need to first find the merger graph or table.

An initial inspection of the compatibility graph does not reveal any subgraph that covers every state of M_8 and consists of just three vertices. In fact, any such graph must contain the subgraph whose vertices are (AC), (BC), (EF), and (CD). Also, since this subgraph is closed, it may seem that there exists no three-state machine that covers M_8. However, it was pointed out earlier that it may be desirable to find a larger closed subgraph if the added vertices can be used to merge compatible pairs to yield larger compatibles. In the above example, if we add the vertex (AB) to the preceding subgraph, we obtain a set that consists of five compatible pairs, $\{(AB), (AC), (BC), (EF), (CD)\}$, and is reducible to the following closed covering:

$$\{(ABC), (CD), (EF)\}.$$

Table 10.19 A minimal machine that covers M_8

		NS, z	
PS		I_1	I_2
(ABC)	α	$\gamma, 0$	$\alpha, 0$
(CD)	β	$\gamma, 1$	$\beta, 0$
(EF)	γ	$\beta, 1$	$\alpha, 0$

Thus, the minimum-state machine that covers M_8 consists of three states and is given in Table 10.19.

Notes and references

The minimization of completely specified machines was first studied by Moore [7] and Huffman [4] and later extended to synchronous machines by Mealy [6]. The reduction procedure for incompletely specified machines is due to Ginsburg [1, 2], Paull and Unger [8], and Kohavi [5]. Other techniques for obtaining minimal machines are available in Grasselli and Luccio [3].

[1] Ginsburg, S.: "A synthesis technique for minimal state sequential machines," *IRE Trans. Electron. Computers*, vol. EC-8, no. 1, pp. 13–24, March 1959.

[2] Ginsburg, S.: "On the reduction of superfluous states in a sequential machine," *J. Assoc. Computing Machinery*, vol. 6, pp. 259–282, April 1959.

[3] Grasselli, A., and F. Luccio: "A method for combined row–column reduction of flow tables," in *Proc. Seventh Symp. Switching and Automata Theory*, Oct. 26–28, pp. 136–147, 1966.

[4] Huffman, D. A.: "The synthesis of sequential switching circuits," *J. Franklin Inst.*, vol. 257, no. 3, pp. 161–190, 1954; no. 4, pp. 275–303, 1954.

[5] Kohavi, Z.: "Minimization of incompletely specified sequential switching circuits," Research Report of the Polytechnic Institute of Brooklyn, PIBMRI, May 1962, New York.

[6] Mealy, G. H.: "A method for synthesizing sequential circuits," *Bell System Tech. J.*, vol. 34, pp. 1045–1079, September 1955.

[7] Moore, E. F.: "Gedanken-experiments on sequential machines," pp. 129–153, in *Automata Studies*, Princeton University Press, 1956.

[8] Paull, M. C., and S. H. Unger: "Minimizing the number of states in incompletely specified sequential switching functions," *IRE Trans. Electron. Computers*, vol. EC-8, pp. 356–366, September 1959.

Problems

Problem 10.1

(a) Prove that $n(n-1)/2$ is an upper bound on the length of the shortest input sequence that will take a strongly connected n-state machine through each of its states at least once, regardless of the initial state. Is this the least upper bound?

(b) Find a one-input 12-state machine for which the length of an input sequence such as that in (a) is as large as possible. (A machine for which the length is 26 can be obtained after a number of trials.)

Problem 10.2. An n-state machine is supplied with a periodic input sequence whose period is p.

(a) Prove that the output sequence must eventually become periodic, and find a bound for the period.

(b) Show the response of the machine M_1^* (Table 10.2) to the input sequence $010010010\cdots$. In particular, find the period of the output sequence and the amount of time required for periodic behavior to start.

Problem 10.3. Prove that there exists no finite-state machine that accepts precisely *all* those sequences that read the same forward as backward, i.e., sequences that are their own reverses. (Such sequences are called *palindromes*.)

Hint: Suppose that there exists an n-state machine that accepts all palindromes; then it accepts the sequence $\underbrace{00\cdots00}_{n+1}1\underbrace{00\cdots00}_{n+1}$. However, this implies that it also accepts a sequence that is not a palindrome.

Problem 10.4. Determine which of the machines with the following specifications is realizable with a finite number of states. If any machine is not realizable, explain why.

(a) A machine is to produce an output symbol 1 whenever the number of 1's in the input sequence, starting at $t = 1$, exceeds the number of 0's. For example, if the input sequence is 01100111, the required output sequence is 00100011.

(b) A machine with a single input line and 10 output lines numbered 0 through 9 is to be designed such that, following the nth input symbol, only one output symbol 1 will be produced on the line whose corresponding number is equal to the nth digit of π (i.e., $3.14\cdots$).

Problem 10.5

(a) Find the equivalence partition for the machine shown in Table P10.5.

(b) Show the *standard form* of the corresponding reduced machine.

(c) Find a minimum-length sequence that distinguishes state A from state B.

Table P10.5

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 1$	$H, 1$
B	$F, 1$	$D, 1$
C	$D, 0$	$E, 1$
D	$C, 0$	$F, 1$
E	$D, 1$	$C, 1$
F	$C, 1$	$C, 1$
G	$C, 1$	$D, 1$
H	$C, 0$	$A, 1$

Problem 10.6. For each machine in Table P10.6, find the equivalence partition and the corresponding reduced machine in standard form.

Table P10.6

	NS, z	
PS	$x=0$	$x=1$
A	$B, 0$	$E, 0$
B	$E, 0$	$D, 0$
C	$D, 1$	$A, 0$
D	$C, 1$	$E, 0$
E	$B, 0$	$D, 0$

(a)

	NS, z	
PS	$x=0$	$x=1$
A	$F, 0$	$B, 1$
B	$G, 0$	$A, 1$
C	$B, 0$	$C, 1$
D	$C, 0$	$B, 1$
E	$D, 0$	$A, 1$
F	$E, 1$	$F, 1$
G	$E, 1$	$G, 1$

(b)

	NS, z	
PS	$x=0$	$x=1$
A	$D, 0$	$H, 1$
B	$F, 1$	$C, 1$
C	$D, 0$	$F, 1$
D	$C, 0$	$E, 1$
E	$C, 1$	$D, 1$
F	$D, 1$	$D, 1$
G	$D, 1$	$C, 1$
H	$B, 1$	$A, 1$

(c)

Problem 10.7. Two columns of the state table of an eight-state p-input symbol finite-state machine are shown in Table P10.7. Prove that this machine has either no equivalent states or no distinguishable states.

Table P10.7

	NS, z		
PS	$\cdots$	I_i	I_j $\cdots$
A		$A, 1$	$H, 0$
B		$C, 1$	$A, 0$
C		$D, 1$	$B, 0$
D		$E, 1$	$C, 0$
E		$F, 1$	$D, 0$
F		$G, 1$	$E, 0$
G		$H, 1$	$F, 0$
H		$B, 1$	$G, 0$

Problem 10.8. A *transfer sequence* $T(S_i, S_j)$ is defined as the shortest input sequence that takes a machine from state S_i to state S_j.

Table P10.8

	NS, z	
PS	$x=0$	$x=1$
A	$A, 0$	$B, 0$
B	$C, 0$	$D, 1$
C	$E, 0$	$D, 0$
D	$F, 0$	$E, 1$
E	$G, 0$	$A, 0$
F	$G, 0$	$B, 1$
G	$C, 0$	$F, 0$

(a) Find a general procedure to determine the transfer sequence for a given machine and two specified states.
(b) Find a transfer sequence $T(A, G)$ for the machine shown in Table P10.8.

Hint: It is helpful to determine which states can be reached from S_i first by sequences of length 1, then by sequences of length 2, and so on.

Problem 10.9

(a) Develop a procedure to determine the shortest input sequence that distinguishes a state S_i from another state S_j of a given machine.
(b) Apply your procedure to determine the shortest input sequence that distinguishes state A from state G in the machine of Table P10.8.

Hint: Start from the first partition P_k in which S_i and S_j appear in separate blocks.

Problem 10.10. The *direct sum* $M_1 + M_2$ of two machines M_1 and M_2 is obtained by combining the tables of the individual machines, as shown in Table P10.10, in such a way that each state of the direct sum is denoted by a distinct symbol.

(a) Use the direct sum to determine whether state A of machine M_1 is equivalent to state H of machine M_2.
(b) Prove that machine M_1 is contained in machine M_2.
(c) Under what starting conditions are machines M_1 and M_2 equivalent?

Hint: Find the equivalence partition of the direct sum.

Table P10.10

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 0$	$C, 1$
B	$D, 1$	$C, 0$
C	$A, 1$	$C, 0$
D	$B, 1$	$C, 0$

M_1

	NS, z	
PS	$x = 0$	$x = 1$
E	$H, 1$	$E, 0$
F	$F, 1$	$E, 0$
G	$E, 0$	$G, 1$
H	$F, 0$	$E, 1$

M_2

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 0$	$C, 1$
B	$D, 1$	$C, 0$
C	$A, 1$	$C, 0$
D	$B, 1$	$C, 0$
E	$H, 1$	$E, 0$
F	$F, 1$	$E, 0$
G	$E, 0$	$G, 1$
H	$F, 0$	$E, 1$

$M_1 + M_2$

Problem 10.11

(a) Let M_1 and M_2 be strongly connected and completely specified machines, and suppose that a state S_i of M_1 is equivalent to a state S_j of M_2. Prove that M_1 is equivalent to M_2.
(b) Let M_1 be a strongly connected machine, and let M_2 be completely specified. Prove that if S_i of M_1 is equivalent to S_j of M_2 then M_1 is covered by M_2.

Problem 10.12. Determine the conditions under which two equivalent machines are isomorphic.

Problem 10.13. An unknown one-input three-state machine produces an output sequence Z in response to an input sequence X, as follows.

X:	0	0	0	0	1	0	1	0	0	0	1	0
Z:	1	0	1	0	0	1	1	0	0	0	0	1

Assuming that A is the initial state, determine the reduced standard form description of the machine.

Problem 10.14. In this problem, we shall establish a procedure for transforming a Mealy machine into a corresponding Moore machine accepting exactly the same set of sequences. To obtain the Moore machine, it is first necessary to split every state of the Mealy machine if different output values are associated with the transitions into that state. For example, state B of Table P10.14a can be reached from either state A or C. However, since different output symbols are associated with these transitions, state B must be replaced by two equivalent states, B_0 with an output symbol 0 and B_1 with an output symbol 1, as shown in Table P10.14b. Every transition to B with output symbol 0 is directed to B_0, and every transition to B with output symbol 1 to B_1. Applying the same procedure to state D yields the state table of Table P10.14b, which can be transformed to the Moore machine of Table P10.14c.

We now observe that the Moore machine in Table P10.14c accepts the sequences accepted by the Mealy machine in Table P10.14a, but, in addition, it produces an output symbol 1 when started in state A without having been presented with any input sequence. This Moore machine in fact accepts a zero-length sequence, called the *null sequence*. To prevent this situation we add a new starting state A', whose state transitions are identical to those of A but whose output symbol is 0, as shown in Table P10.14d.

(a) Prove that, to every q-output-symbol n-state Mealy machine, there corresponds a q-output-symbol Moore machine that accepts exactly the same sequences and has no more than $qn + 1$ states.

(b) If the definition of acceptance by a Moore machine is modified so that acceptance of the null sequence is disregarded, show a procedure for transforming a Moore machine to the corresponding Mealy machine such that both accept the same set of sequences.

(c) Prove that if the Mealy machine is strongly connected and completely specified, the corresponding Moore machine will also be strongly connected and completely specified.

Table P10.14

	NS, z	
PS	$x = 0$	$x = 1$
A	$C, 0$	$B, 0$
B	$A, 1$	$D, 0$
C	$B, 1$	$A, 1$
D	$D, 1$	$C, 0$

(a)

	NS, z	
PS	$x = 0$	$x = 1$
A	$C, 0$	$B_0, 0$
B_0	$A, 1$	$D_0, 0$
B_1	$A, 1$	$D_0, 0$
C	$B_1, 1$	$A, 1$
D_0	$D_1, 1$	$C, 0$
D_1	$D_1, 1$	$C, 0$

(b)

	NS		
PS	$x=0$	$x=1$	z
A	C	B_0	1
B_0	A	D_0	0
B_1	A	D_0	1
C	B_1	A	0
D_0	D_1	C	0
D_1	D_1	C	1

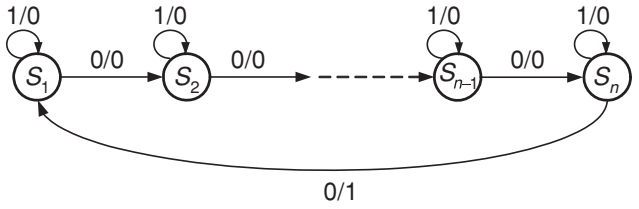
(c)

	NS		
PS	$x=0$	$x=1$	z
A'	C	B_0	0
A	C	B_0	1
B_0	A	D_0	0
B_1	A	D_0	1
C	B_1	A	0
D_0	D_1	C	0
D_1	D_1	C	1

(d)

Problem 10.15. By referring to the machine shown in Fig. P10.15, prove that the bound established in Theorem 10.2 is the least upper bound; that is, show that, for every n, the states in the pair $(S_1 S_2)$ cannot be distinguished by a sequence shorter than $n-1$.

Fig. P10.15



Problem 10.16. A given machine is known to be either a machine M_1 in state S_i or a machine M_2 in state S_j, where S_i is not equivalent to S_j. Suppose that you are given the state tables of M_1 and M_2, and assume that M_1 has n_1 states and M_2 has n_2 states. Prove that the given machine and its initial state can always be identified by means of an input sequence whose length L is bounded by $L \leq n_1 + n_2 - 1$.

Problem 10.17. Give a procedure that can be used to determine whether two incompletely specified machines M_1 and M_2 are related, in such a way that either M_1 contains M_2 or vice versa.

Problem 10.18

(a) Find all the state containments present in the machine shown in Table P10.18.

(b) Find two minimum-state machines that contain the given machine, and prove that these machines are indeed minimal.

Table P10.18

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 0$	$C, 1$
B	$D, 0$	$C, 1$
C	$A, 0$	$E, 0$
D	—	$F, 1$
E	$G, 1$	$F, 0$
F	$B, 0$	—
G	$D, 0$	$E, 0$

Problem 10.19. For the incompletely specified machines shown in Table P10.19, find a minimum-state reduced machine containing the original one.

Table P10.19

	NS, z		
PS	I_1	I_2	I_3
A	$C, 0$	$E, 1$	—
B	$C, 0$	$E, -$	—
C	$B, -$	$C, 0$	$A, -$
D	$B, 0$	$C, -$	$E, -$
E	—	$E, 0$	$A, -$

	NS, z	
PS	I_1	I_2
A	—	$F, 0$
B	$B, 0$	$C, 0$
C	$E, 0$	$A, 1$
D	$B, 0$	$D, 0$
E	$F, 1$	$D, 0$
F	$A, 0$	—

Problem 10.20. Prove that the machine shown in Table P10.20 is minimal.

Table P10.20

	NS, z						
PS	I_1	I_2	I_3	I_4	I_5	I_6	I_7
A	$F, 0$	$A, -$	$D, -$	$C, -$	—	—	—
B	$-, 1$	—	—	—	$C, -$	$D, -$	$E, -$
C	$C, -$	$E, -$	—	—	$F, 0$	$B, -$	—
D	—	—	$F, -$	$E, -$	$-, 1$	—	$A, -$
E	$A, -$	—	$A, 1$	—	$B, -$	—	$C, -$
F	—	$D, -$	$-, 0$	$B, -$	—	$E, -$	—

Problem 10.21. Find the reduced state table for the machine of Table P10.21. Design the circuit using a single SR flip-flop.

Table P10.21

	NS, z_1z_2			
PS	00	01	11	10
A	A, 00	E, 01	—	A, 01
B	—	C, 10	B, 00	D, 11
C	A, 00	C, 10	—	—
D	A, 00	—	—	D, 11
E	—	E, 01	F, 00	—
F	—	G, 10	F, 00	G, 11
G	A, 00	—	—	G, 11

Problem 10.22. Design a serial-to-parallel Excess-3-to-BCD code converter. The circuit has a single input line, receiving messages in Excess-3 code, and four output lines, z_1, z_2, z_4, and z_8, which are to reproduce the input messages in BCD code. Input symbols arrive serially, with the least significant digit first. Output symbols are specified only at the occurrence of every fourth input symbol. For example, if the input sequence is 1001 (which is 6 in Excess-3 code), the required output sequence is $z_1 = 0$, $z_2 = 1$, $z_4 = 1$, $z_8 = 0$.

CHAPTER

11 Asynchronous sequential circuits

In many practical situations, synchronous circuits lead to more power consumption and delay than asynchronous circuits. Moreover, within large synchronous systems, it is often desirable to allow certain subsystems to operate asynchronously, thereby avoiding some of the problems associated with clocking. In this chapter, we present some of the basic properties of asynchronous sequential circuits and methods for their synthesis.

11.1 Modes of operation

Although there are many forms that an asynchronous sequential circuit might take, the one shown in Fig. 11.1 is the most straightforward for a quick understanding of how such a circuit works. Externally, the circuit is characterized by the fact that its inputs can change at any time. Internally, it is characterized by the use of delay elements as memory devices.[1]

The combination of the signals that appear at the primary inputs and delay outputs defines what is called the *total state* of the circuit. The combination of input signals $x_1, x_2, \ldots, x_l$ is referred to as the *input state*; the combination of signals at the outputs of the delays, i.e., $y_1, y_2, \ldots, y_k$, is referred to as the *secondary* or *internal state* of the circuit. The output values generated by the combinational logic define the output symbol of the entire circuit as well as the secondary state that the circuit will assume next. The variables $y_1, y_2, \ldots, y_k$ are referred to as *secondary* or *internal variables*, and the variables $Y_1, Y_2, \ldots, Y_k$ are called *excitation variables*.

For a given input state, the circuit is said to be in a *stable state* if and only if $y_i = Y_i$ for $i = 1, 2, \ldots, k$. In response to a change in the input state,

[1] In practice, when the inherent delay of the combinational logic is large enough the external delay elements may not be necessary. However, for clarity of presentation, we shall assume they are present.

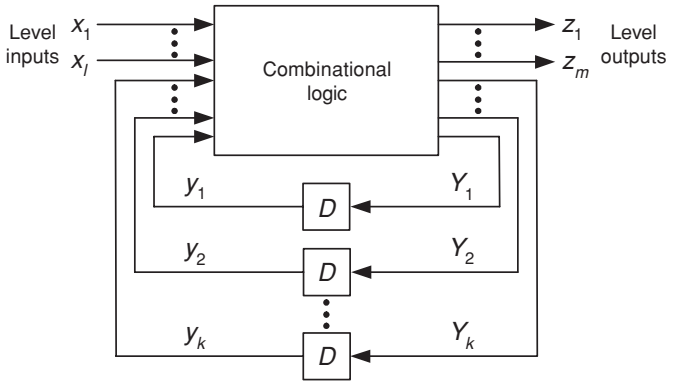
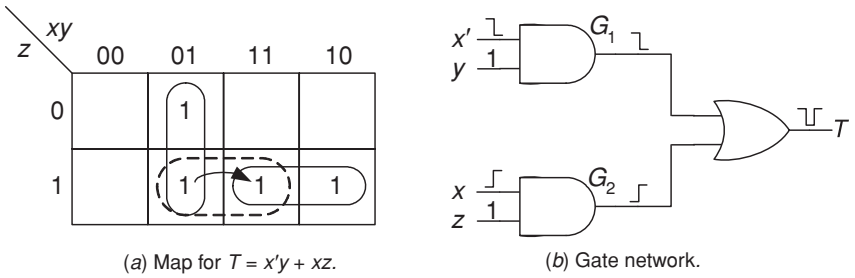


Fig. 11.1 The basic model for fundamental-mode circuits.

the combinational logic produces a new set of values for the excitation variables. As a result, the circuit enters what is called an *unstable state*. When the secondary variables assume their new values, i.e., the y's become equal to the corresponding Y's, the circuit enters its "next" stable state. Thus, *a transition from one stable state to another occurs only in response to a change in the input state*. We shall initially assume that, after a change in input values has occurred, there is no other change in any input value until the circuit enters a stable state. Such a mode of operation is often referred to as the *fundamental mode*. If only a single input value is allowed to change at any given time, it is called a *single-input-change* (*SIC*) *fundamental mode*, otherwise a *multiple-input-change* (*MIC*) *fundamental mode*. Even though SIC fundamental-mode circuits work under very restrictive assumptions, we will discuss first methodologies applicable to them, for ease of exposition, and then those applicable to MIC fundamental-mode circuits. We will then consider a generalization of MIC fundamental-mode circuits called *burst-mode* circuits. There are many other types of asynchronous circuits as well. However, they are beyond the scope of this book.

11.2 Hazards

Hazards refer to glitches. They are of two types: logic hazards and function hazards. *Logic hazards* are caused by noninstantaneous changes in circuit signals. *Function hazards* are inherent in the functional specification. The presence of hazards poses a fundamental challenge to the design of asynchronous circuits since a glitch may be misunderstood by another part of the circuit as a valid transition and cause incorrect behavior. Since we are interested in both SIC and MIC fundamental modes of operation, we will see how hazards can form under each mode and how to design circuits to be free of hazards whenever possible.



(*a*) Map for $T = x'y + xz$. (*b*) Gate network.

Fig. 11.2 Single-input-change static hazard example.

Design of SIC hazard-free circuits

Consider the function $T(x, y, z) = \sum(2, 3, 5, 7)$, whose map is shown in Fig. 11.2*a*, and its minimal sum-of-products implementation in Fig. 11.2*b*. Suppose that the value of inputs y and z is 1 and that the value of input x is changed from 0 to 1. Clearly, the value of T must remain at 1 regardless of the value of x. As the value of x changes, the transmission path through the network of Fig. 11.2*b* changes from gate G_1 to G_2. In an ideal situation this change would be instantaneous, and the value of T would remain constant at 1. In practice, however, different delays are associated with the gates G_1 and G_2. As a consequence, if, for example, the delay of gate G_1 is smaller than that of gate G_2, and if x changes from 0 to 1 (while $y = z = 1$), then the transmission $x'y$ through gate G_1 will become 0 shortly before the transmission xz through gate G_2 becomes 1. During this period, T will be 0. This phenomenon is known as a *static logic hazard* and is indicated by the arrow in the map of Fig. 11.2*a*. More specifically, since only a single bit changes in the transition, it is called an *SIC static logic hazard*. In general, an SIC static logic hazard is a scenario in which a single input-variable change might cause a momentarily incorrect output value when, in fact, the output value should remain constant. Whether such an incorrect output value actually occurs depends on the exact amounts of delay associated with the various circuit elements.

Two input combinations are said to be *adjacent* if they differ by the value of a single input variable. For example, $x'yz$ and xyz are adjacent. A transition between a pair of adjacent input combinations that correspond to identical output values contains an SIC static logic hazard if it makes possible the generation of a momentary spurious output value. Such hazards may occur whenever there exists a pair of adjacent input combinations that produce the same output value and there is no cube (in the map) containing both combinations.

On the basis of the above discussion, in the above example the static logic hazard can be removed by including the prime implicant yz in the expression for T, as indicated by the dotted cube in the map of Fig. 11.2*a*, that is, writing $T = x'y + xz + yz$. The resulting circuit is shown in Fig. 11.3. Clearly, when

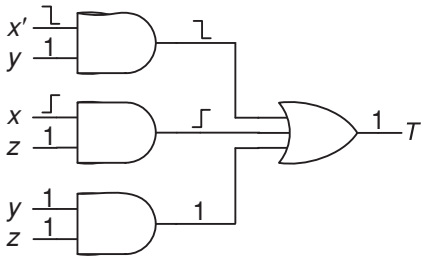


Fig. 11.3 Single-input change hazard-free network.

$y = z = 1$ the output value will be 1 regardless of the delays associated with x' and x.

When the hazard occurs during a static $0 \rightarrow 0$ transition at the output it is called a *static-0 logic hazard*, and for a $1 \rightarrow 1$ transition a *static-1 logic hazard*.

A *transition cube* $[m_1, m_2]$ is a set of all minterms that can be reached starting from minterm m_1 and ending at minterm m_2. For example, the transition cube [010, 100] contains the following minterms: 000, 010, 100, 110. In the example in Fig. 11.2, we saw that transition cube [011, 111] must be included in some product of the sum-of-products realization in order to get rid of the static-1 logic hazard. Such a cube is called a *required cube*.

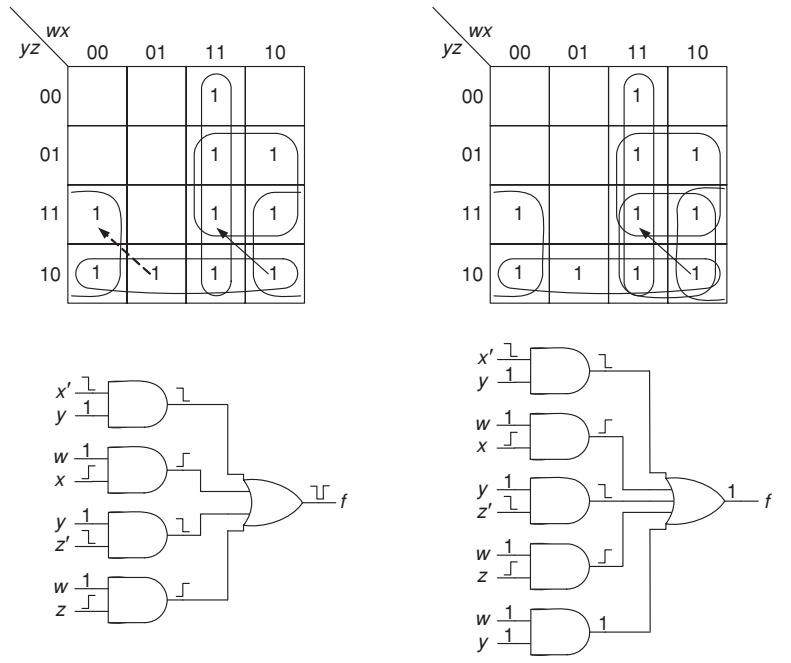
In the sum-of-products realization of a function, no cube for any product term can contain either of the two input combinations involved in a $0 \rightarrow 0$ output transition since a cube only includes the 1's of a function. Thus, the only way in which a static-0 logic hazard can occur is if a product term has both x_i and x_i' as input literals. Since there is no reason to include such product terms in the expression for the function, such hazards can be trivially avoided.

If the two input combinations are such that they correspond to a $0 \rightarrow 1$ output transition but during the transition the 0 may change to 1 and then 0 and finally stabilize at 1 then the sum-of-products realization is said to have a *dynamic* $0 \rightarrow 1$ *logic hazard*. A dynamic $1 \rightarrow 0$ logic hazard can be similarly defined. Using reasoning similar to that above for static-0 logic hazards, a dynamic $0 \rightarrow 1$ or $1 \rightarrow 0$ logic hazard is not possible in the SIC scenario unless some product term has both x_i and x_i' as input literals.

Design of MIC hazard-free circuits

In an MIC scenario, several inputs change values *monotonically*, i.e., at most once, from one input combination to another. In this transition, if the function changes values more than once then the transition is said to have a *function hazard*.

Example Consider the MIC transition, denoted by the broken arrow in the map shown in Fig. 11.4*a*, from $wxyz = 0110$ to $wxyz = 0011$. If z changes before x does then the function will go from 1 to 0 and then back to 1. Hence, the function changes values more than once and thus this transition has a function hazard.



(*a*) Static-1 logic hazard. (*b*) Logic-hazard-free network.

Fig. 11.4 Static MIC hazard.

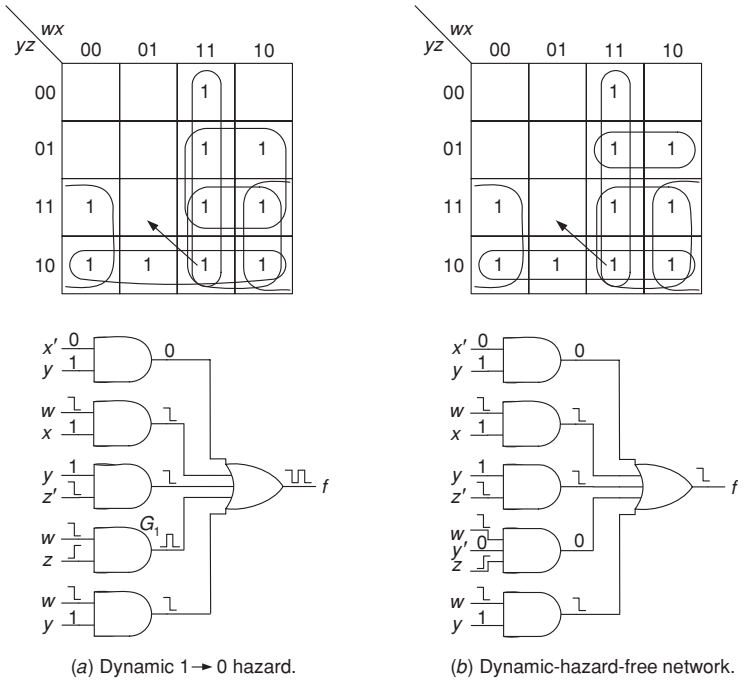
If a transition has a function hazard then no implementation can be guaranteed to be hazard-free for this transition, assuming that the gates and wires have arbitrary delays, because the glitch is present in the functional specification itself. Fortunately synthesis approaches, such as those based on the burst mode, only need to deal with transitions that are free of function hazards. Thus, we shall focus only on MIC transitions that are free of function hazards.

Example Consider the MIC transition, denoted by the solid arrow in the map shown in Fig. 11.4*a*, from $wxyz = 1010$ to $wxyz = 1111$. This transition does not have a function hazard. However, it may lead to a static-1 logic hazard, as shown in the AND–OR circuit in Fig. 11.4*a*. Such a hazard could occur in a situation in which the falling transitions at the outputs of two AND gates are faster than the rising transitions at the outputs of the other two AND gates. Such hazards can be tackled in the same manner as those caused by an SIC transition, as shown in Fig. 11.4*b*. The AND gate that realizes wy has a steady 1 at its output during the above transition. The reason is that it covers the entire required cube [1010, 1111] in the map. Such a cube includes all the minterms that can be encountered during such a monotonic transition. This eliminates the hazard at f.

Just as in the SIC case, avoiding a static-0 logic hazard is straightforward (simply avoid any product term with both x_i and x_i' as literals). Thus, we will look at MIC dynamic hazards next.

Example Consider the MIC transition, denoted by the solid arrow in the map shown in Fig. 11.5*a*, from $wxyz = 1110$ to $wxyz = 0111$. This dynamic transition does not have a function hazard. However, the transition does have a dynamic logic hazard, as can be seen from the AND–OR circuit in Fig. 11.5*a*. This dynamic hazard may be created by a combination of the static-0 hazard at the output of the AND gate G_1 and the falling transition at the outputs of several other AND gates.

A necessary condition for a dynamic transition to be hazard-free is that each of its $1 \rightarrow 1$ subtransitions are also hazard-free. This can be ensured by including these subtransitions in some product of the sum-of-products realization. For the above dynamic transition, these subtransitions are [1110, 1111] and [1110, 0110]. They are called the required cubes of this dynamic transition. The set of required cubes includes all minterms that can be encountered in the dynamic transition. Since [1110, 1111] and [1110, 0110] are included in the products wx and yz', respectively, the above necessary condition is already met in this case.



(*a*) Dynamic 1→ 0 hazard. (*b*) Dynamic-hazard-free network.

Fig. 11.5 Dynamic MIC hazard.

In order to prevent the dynamic hazard at f, we also need to make sure that no AND gate temporarily turns on during the MIC transition. For example, the static-0 hazard at the output of G_1 needs to be avoided. This hazard is caused when G_1 temporarily turns on. This happens because the corresponding product term wz *intersects* the dynamic MIC transition $1110 \rightarrow 0111$. This is called an *illegal intersection* and the dynamic transition is called a *privileged cube*. One can see that, during this dynamic transition, the inputs could be momentarily at 1111 (if z changes before w), which is a minterm of wz. To avoid this situation, illegal intersections of privileged cubes are disallowed by reducing the product term wz to $wy'z$, as shown in the map in Fig. 11.5*b*, thus eliminating the hazards as can be seen from the corresponding circuit.

The above discussions show how to eliminate hazards for an MIC transition. An MIC transition that results in a $1 \rightarrow 1$ transition at the output must be completely covered by a product term. The $0 \rightarrow 0$ MIC transition does not lead to a hazard. For the $1 \rightarrow 0$ and $0 \rightarrow 1$ cases, we have to make sure that every product term that intersects the MIC transition also contains its starting or end point, respectively.

To obtain a hazard-free sum-of-products implementation H of function f for a specified set of input transitions, we need to make sure that (i) each required cube is contained in some implicant of H and (ii) no implicant of H *illegally intersects* any specified dynamic transition. Such an implicant is called a dynamic-hazard-free implicant (dhf-implicant).

The above problem requires that we make use only of dhf-prime implicants[2] while covering every required cube in the sum-of-products minimization. This is similar to the Quine–McCluskey minimization method we discussed in Chapter 4.

Example Consider the map shown in Fig. 11.6*a*. It contains four function-hazard-free transitions, depicted by the four arrows. The set of required cubes is also shown in this map. The map in Fig. 11.6*b* shows the set of privileged cubes. The prime implicants that do not have any illegal intersections with the two dynamic transitions ($1101 \rightarrow 0000$ and $0011 \rightarrow 0110$), i.e., the dhf-prime implicants, are shown in Fig. 11.6*c*. However, the prime implicant xz does have an illegal intersection with the transition $0011 \rightarrow 0110$, as shown by the shaded region in Fig. 11.6*d*. This intersection can be avoided by reducing xz to the dhf-prime implicant $xy'z$, as shown in Fig. 11.6*e*.

[2] A *dhf-prime implicant* is a dhf-implicant that is not contained in any other dhf-implicant.

Table 11.1 Chart for dhf-prime implicants

Dhf-prime implicants	Required cubes				
	wy'	wy	$xy'z$	$w'yz$	$w'x'y$
w	×	×			
yz				×	
$x'y$					×
$xy'z$			×		

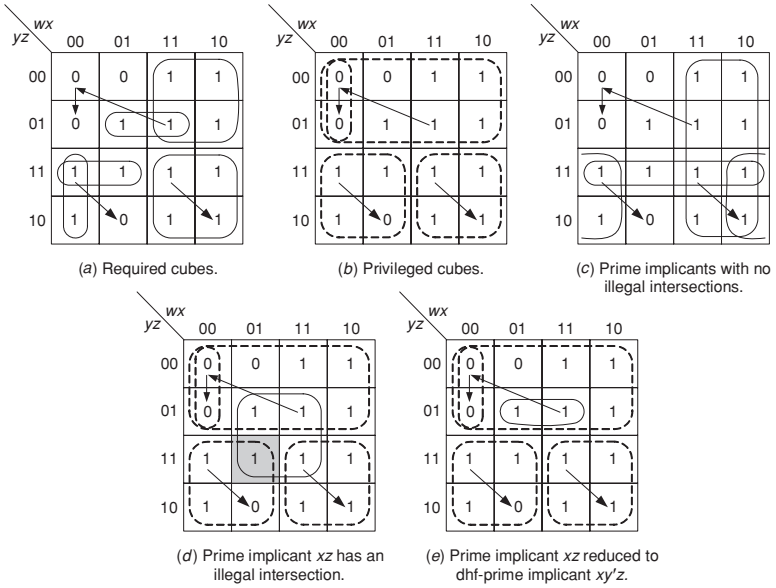


Fig. 11.6 Derivation of a hazard-free sum-of-products expression.

A minimal hazard-free sum-of-products realization can now be obtained using a concept similar to the prime implicant chart (see Chapter 4). This is shown in Table 11.1, in which the rows correspond to dhf-prime implicants and the columns to required cubes. The aim is to find a minimal set of dhf-prime implicants that contains all required cubes. This can be done using the analogous concepts of essential rows, dominated rows, and dominating columns used earlier for prime implicant charts. From Table 11.1, we see that all rows are essential. Thus, $w + yz + x'y + xy'z$ is a hazard-free sum-of-products expression.

Since the hazard-free AND–OR implementation may be too large, it may be necessary to obtain a hazard-free multi-level implementation from it. In order to do so, we have to apply *hazard-nonincreasing* logic transformations. These transformations ensure that if the initial circuit is hazard-free, so is the final circuit. The following laws from Boolean algebra constitute some hazard-nonincreasing transformations:

- the associative law, $(x + y) + z \Leftrightarrow x + (y + z)$, and its dual, $(xy)z \Leftrightarrow x(yz)$,
- De Morgan's theorem, $(x + y)' \Leftrightarrow x'y'$, and its dual, $(xy)' \Leftrightarrow x' + y'$,
- the distributive law, $xy + xz \Rightarrow x(y + z)$,
- the absorption law, $x + xy \Rightarrow x$, and
- the $x + x'y \Rightarrow x + y$ law.

The directions of the implication arrows indicate in which directions the transformations are applicable. Similarly, the insertion of inverters at the primary inputs and the circuit output is also hazard-nonincreasing.

Example Consider the AND–OR realization in Fig. 11.5*b*, which is dynamic-hazard-free for the MIC transition $1110 \rightarrow 0111$. A multi-level realization can be obtained from it using the distributive law: $x'y + wx + yz' + wy'z + wy = (x' + z' + w)y + wx + wy'z$, as shown in Fig. 11.7. As can be seen, this multi-level realization is also dynamic-hazard-free.

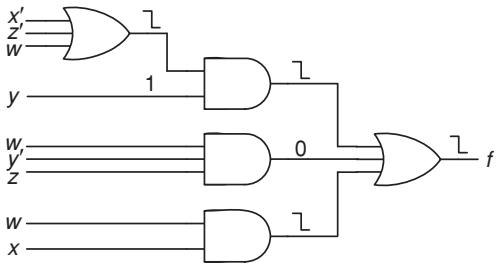


Fig. 11.7 Multi-level hazard-nonincreasing realization.

11.3 Synthesis of SIC fundamental-mode circuits

The purpose of this section is to develop systematic techniques for the design of SIC fundamental-mode asynchronous sequential circuits. The approach to be followed is to construct a flow table which describes the circuit behavior, to simplify the table, whenever possible, and finally, to realize it at the gate level.

The flow table

As in the case of synchronous circuits, the least systematic step in the synthesis procedure is that of transforming a verbal statement of the desired circuit behavior into a precise description that specifies the circuit operation for every applicable input sequence. A convenient method for describing the behavior of an asynchronous circuit is by means of a *flow table*. As an example, consider a sequential circuit with two inputs, x_1 and x_2, and one output, z. The initial

Table 11.2 Partial flow table

	State, output			
x_1x_2				
00	01		11	10
1, 0 →	2			
	↓			
	2, 0	→	3	
			↓	
			3, 1	

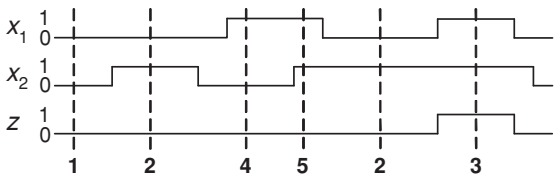


Fig. 11.8 Input–output sequences.

input state is $x_1 = x_2 = 0$. The output value is to be 1 if and only if the input state is $x_1 = x_2 = 1$ and the preceding input state is $x_1 = 0$, $x_2 = 1$. A possible pair of input sequences and the corresponding output sequence are illustrated in Fig. 11.8.

We now show how to construct the flow table for the given circuit. The column headings of Table 11.2 are the input combinations. The table entries give the states and output values. The arrows indicate state transitions between the table entries. Initially, the input values are $x_1 = x_2 = 0$, and the circuit is in a state designated **1**; the use of boldface indicates that the state in question is stable. This is recorded in the table by entering a **1** in the first row of column $x_1x_2 = 00$. To the right of the **1**, output entry 0 is entered, since the output value is 0 when the circuit is in state **1**. Now x_2 becomes 1 while x_1 remains 0, as illustrated in Fig. 11.8; the circuit enters a different state, designated **2**, while its output value is still 0. This is recorded in Table 11.2 by entering a **2** in the second row, column $x_1x_2 = 01$, and a 0 in the corresponding output location. In the first row of the 01 column, we enter a 2 to indicate that, as a result of the change in the value of the input variables, a transition to state **2** will occur. Thus, while the lightface entry 2 designates an *unstable* transient condition, the boldface entry **2** designates the *stable* state assumed by the circuit as a result of the above input change. If input x_1 changes from 0 to 1 while the circuit is in state **2**, the circuit enters another stable state, designated **3**, which is associated with the output value $z = 1$. This is indicated by entering a lightface 3 in the second row, column 11. In the same column and immediately below the lightface 3, a boldface **3** is entered to identify the stable state to which the circuit goes as a result of the last change of input values. The output value 1 is associated with the stable state **3**.

Table 11.3 Primitive flow table

State, output			
x_1x_2			
00	01	11	10
1, 0	2	—	4
1	**2**, 0	3	—
—	2	**3**, 1	4
1	—	5	**4**, 0
—	2	**5**, 0	4

Thus a change in the value of the circuit inputs causes a *horizontal* move in the flow table to the column whose heading corresponds to the new input value. A change in the internal state of the circuit is reflected by a *vertical* move, as shown by the arrows in Table 11.2. (Note that, since a change in the inputs can occur only when the circuit is in a stable state, a horizontal move can emanate only from a boldface entry.) For the time being, we shall specify only the output symbols of stable states, leaving the output symbols of unstable states for later consideration.

So far, we have specified the state transitions leading from the initial state to a state that generates an output value 1. Clearly, we must also specify what is to happen if an input sequence other than the one considered occurs. Suppose, for example, that initially x_1 changes before x_2. As a result, the circuit will go through unstable state 4 to stable state **4** (see Table 11.3), for which the output symbol is 0. Since the two inputs are not allowed to change simultaneously, a dash is entered in the first row, column 11, and in the second row, column 10, of Table 11.3 and so on. In general, to specify the operation of a circuit, we use a partly developed table similar to Table 11.2 and specify the transitions for each allowable input change, starting from every stable state. If a new stable state is to be added, a *new* row is created in the column corresponding to the present values of input variables. Any move from a stable state can be caused only by a change in the input variables.

The table thus constructed is called *a primitive flow table*. Its main characteristics are that *only one stable state appears in each row and the output symbols are specified only for stable states*.

We will now complete the flow table. Starting from entry **2** in column 01, if the inputs change to 00, it is necessary to send the circuit into the state that corresponds to the input conditions $x_1 = x_2 = 0$ and output $z = 0$, i.e., the state **1**. Therefore, a lightface 1 is entered in column 00 of the row containing **2**. The circuit can leave state **3** by a change of inputs from $x_1x_2 = 11$ to either $x_1x_2 = 01$ or $x_1x_2 = 10$. In the first case the value of input x_1 has changed from 1 to 0, while x_2 remains equal to 1; if x_1 changes again (to 11, we want the circuit to go back to state **3** and to produce a 1 output value.

This transition can be accomplished if we enter a lightface 2 in column 01 in the third row. If, however, x_2 changes from 1 to 0 while x_1 remains at 1 then the circuit goes to state **4**, which satisfies these conditions. Starting from state **4**, we observe that if the value of x_2 changes from 0 to 1 then the two circuit input values are 1's. However, since the last input to change was x_2, not x_1, the output value should be 0. Consequently, a new state, designated **5**, for which the output value is 0 must be added in column 11.

At this point, we have obtained all the stable states shown in Table 11.3. The table is completed by entering the unstable states corresponding to the various possible changes of input variables. A dash has been entered wherever a change of input variables is not allowed.

Reduction of flow tables

The primitive flow table developed in Table 11.3 has five *distinct* states. Thus, it appears that at least three variables are needed to represent these states. However, as we shall see, this does not necessarily mean that three secondary variables must be employed, since the input variables may be used to distinguish some of the states. This problem can be better understood if we think of each *stable state* as representing a *total state* of the circuit, i.e., a state defined by the state of the internal (i.e., secondary) variables as well as by the state of the primary input variables. Accordingly, an asynchronous circuit can go from one stable state to another stable state without necessarily changing the values of any of its internal variables. Such a situation simply means that these two states are distinguished only by the states of the input variables. (Note that in the case of synchronous circuits the input variables cannot be used to specify the total state of the circuit since, although a synchronous circuit is stable when the clock pulses are absent, the input values are not available to it.)

In general, when setting up a primitive flow table, one is not concerned about adding states that may turn out to be redundant. All that is necessary is that a sufficient number of states be included, such that the circuit behavior is completely specified for every allowable input sequence. The reduction of a primitive flow table thus has two functions, namely, eliminating redundant stable states and merging those stable states that are distinguishable by the input states. Since there is only one stable state in each row of the primitive flow table, we may think of it as the "present state (PS)" and rewrite Table 11.3 in the form shown in Table 11.4, where the boldface entries again serve to identify stable states. The flow table in the form of Table 11.4 is now indistinguishable from the state table of an incompletely specified synchronous circuit, possibly with the exception that every row of the flow table contains one "next-state" entry which is identical to the "present state."

The analogy between the minimization problem of synchronous circuits and the reduction of primitive flow tables of asynchronous circuits is now apparent.

Table 11.4 Primitive flow table

	State, output			
PS	x_1x_2 00	01	11	10
1	**1**, 0	2	—	4
2	1	**2**, 0	3	—
3	—	2	**3**, 1	4
4	1	—	5	**4**, 0
5	—	2	**5**, 0	4

Table 11.5 Reduced flow tables

State, output			
x_1x_2 00	01	11	10
1, 0	**2**, 0	**3**, 1	4, 0
1, 0	2, 0	**5**, 0	**4**, 0

(*a*) The closed covering {(123), (45)}

State, output			
x_1x_2 00	01	11	10
1, 0	2, 0	**5**, 0	**4**, 0
1, 0	**2**, 0	**3**, 1	4, 0

(*b*) The closed covering {(145), (23)}

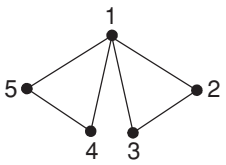


Fig. 11.9 Merger graph for Table 11.4.

We may, therefore, utilize the techniques of Section 10.4 to reduce the number of rows in primitive flow tables. The merger graph for the flow table of Table 11.4 is shown in Fig. 11.9, where the maximal compatibles are {(123), (145)}. Whenever bold and lightface entries are to be combined, the resulting entry is bold since the corresponding state must be stable. Thus, for example, the row of Table 11.4 that corresponds to the maximal compatible (123) is

$$\mathbf{1}, 0 \quad \mathbf{2}, 0 \quad \mathbf{3}, 1 \quad 4, -$$

Two minimum-row flow tables corresponding to Table 11.3 are shown in Table 11.5. Table 11.5*a* corresponds to the closed covering {(123), (45)} while Table 11.5*b* corresponds to the closed covering {(145), (23)}. The output symbols associated with unstable states have been specified to correspond to their respective stable states, e.g., the output symbol associated with the unstable state 2 is 0 since the output symbol of the stable state **2** is 0, and so on.

Specifying the output symbols

Our next step is to consider the assignment of output values to the unstable states in the reduced flow table. This assignment depends on the required output value changes, as well as on a number of design objectives that will be discussed subsequently. Suppose that the circuit is to go from one stable state to another stable state associated with the same output value, as is the case, for example,

Table 11.6 Specification of output symbols

State, output			
x_1x_2			
00	01	11	10
1, 0	2	**3**, 0	4
1	**2**, 1	3	**4**, 0
5, 1	6	**7**, 1	8
5	**6**, 0	7	**8**, 0

(*a*) Reduced flow table

State, output			
x_1x_2			
00	01	11	10
1, 0	2, 1	**3**, 0	4, 0
1, 0	**2**, 1	3, 0	**4**, 0
5, 1	6, 0	**7**, 1	8, 0
5, 1	**6**, 0	7, 1	**8**, 0

(*b*) Reduced flow table with output values specified

in Table 11.5*a* in the transition from state **1** to state **4**. In such a case there must be no momentary complementary output value. Consequently, unstable state 4 must be assigned a 0 output value. Similarly, the output value associated with unstable state 2 is specified as 0.

When a circuit changes from one stable state with a given output value to another stable state with a different output value, the transition may be associated with either output value. The choice of output value can be made according to whether it is desired that the output-value change will occur as soon as possible or as late as possible. When the relative timing of the output-value change is of no importance, the choice of output value is made in such a way as to minimize the output logic. Consider, for example, the flow table in Table 11.6*a*. To determine the output value associated with unstable state 2, note that state **2** can be reached from either state **1** or state **3**. Since both are associated with a 0 output value, while the output value of state **2** is 1, then if a fast output value change is desired the output value associated with unstable state 2 must be a 1 but if a slow output value change is desired then the output of 2 should be set to 0. However, the output of unstable state 1 must be set to 0, since the output values of states **1** and **4** are both 0's.

The output value associated with unstable state 4 must be a 0, as must the output value associated with 3, since in each case the transition is between stable states associated with 0 output values. Note that this output assignment means that the output value associated with the transition from **2** to **3** cannot be made in such a way that the change is as late as possible. An examination of the output values associated with the unstable states in the last two rows shows that they are all optional. The output assignment shown in Table 11.6*b* has been made in such a way as to obtain fast output value changes.

Excitation and output tables

To realize a reduced flow table, it is necessary to assign distinct combinations of the secondary-variable values to the rows of the flow table and derive the corresponding excitation and output functions. For a state to be stable, the

Table 11.7 Excitation and output table

	Y, z			
	x_1x_2			
y	00	01	11	10
0	0, 0	0, 0	0, 1	1, 0
1	0, 0	0, 0	1, 0	1, 0

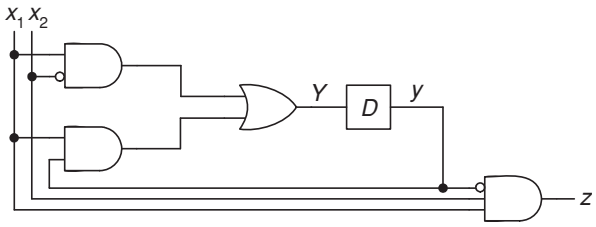


Fig. 11.10 A realization of Table 11.7.

values of the Y's must be the same as those of the y's. Therefore, the excitation required for any stable state is determined from the value of the secondary variables assigned to the row in which the stable state is contained. A lightface entry represents an unstable state, which must eventually assume the value of the secondary state assigned to the boldface entry having the same number. There are several difficulties associated with the state-assignment problem and with the transitions assigned to the unstable states. These problems are discussed in detail later.

To realize the reduced flow table of Table 11.5*a*, we assign a 0 to the first row and a 1 to the second row, as shown in Table 11.7. Every boldface entry in the first row is now replaced by a 0, and in the second row by a 1. The lightface entry 2 is assigned a 0, since the circuit must go into stable state **2**; this assignment thus requires the variable y to change its state from 1 to 0 upon receiving input symbol 01. Similarly, the lightface entries 1 and 4 are assigned 0 and 1, corresponding respectively to the assignments of the boldface entries **1** and **4**. The excitation and output functions derived from Table 11.7 are

$$Y = x_1x_2' + x_1y,$$
$$z = x_1x_2y'.$$

A corresponding realization is shown in Fig. 11.10.

A synthesis example

The synthesis procedure for SIC fundamental-mode asynchronous circuits developed in the foregoing section consists of several steps, which can be summarized as follows.

1. A primitive flow table is constructed from the verbal description of circuit operation. In most cases, we specify only those output values that are associated with stable states.
2. A minimum-row reduced flow table is obtained by merging the rows in the primitive flow table. Either the merger graph or the merger table may be used to perform the reduction.
3. Secondary variables are assigned to the rows of the reduced flow table, from which excitation and output tables are constructed. The output values associated with unstable states are specified according to various design requirements.
4. The excitation and output functions are derived, and the corresponding hazard-free circuit constructed.

We shall now illustrate the above procedure by designing an asynchronous sequential circuit with two inputs, x_1 and x_2, and two outputs, G and R, which is to behave in the following manner. Initially, both input values and both output values are equal to 0. Whenever $G = 0$ and either the value of x_1 or x_2 becomes 1, G turns "on" (i.e., attains the value 1). When the value of the second input becomes 1, R turns on. The first input value that changes from 1 to 0 turns G "off" (i.e., sets G equal to 0). The output R turns off when G is off and either input value changes from 1 to 0.

From the specification of the problem, it is evident that whenever $x_1 = x_2 = 0$ then $G = R = 0$, and whenever $x_1 = x_2 = 1$ then $G = R = 1$. Consequently, columns 00 and 11 of the primitive flow table must each contain a single stable state. When the input combination x_1x_2 is 01, the output symbol GR may be either 10 or 01, depending on the preceding input combination. Since a different stable state must be included in each column of the flow table for every possible output condition, column 01 must contain at least two stable states. Similar arguments show that column 10 must also contain at least two stable states, which will be associated with the output combinations 01 and 10. We thus conclude that the primitive flow table for the circuit in question must contain six stable states, as illustrated in Table 11.8*a*. The primitive flow table can now be completed by inserting the dashes, whenever a multiple change of input values is implied, and by specifying the unstable states.

When the circuit is in state **1**, any allowed change of input symbols causes a change in output symbols from 00 to 10. Hence, the circuit must be directed to either state **2** or **5**, depending on whether the change in input symbols is from 00 to 01 or 10, respectively. This is accomplished by entering a 2 in column 01 and a 5 in column 10 in the first row of Table 11.8*b*. It is a simple matter to complete the unstable entries in columns 00 and 11, since each of these columns contains just a single stable state. Therefore, 1's and 4's are entered in the appropriate locations in Table 11.8*b*. The only as yet unspecified entries are those in the row containing **4** in columns 01 and 10. If we start from state **4** and change the input symbols to 01 or 10, G must be turned off. Hence, we

Table 11.8 Primitive flow table

State, GR			
x_1x_2 00	01	11	10
1, 00			
	2, 10		
	3, 01		
		4, 11	
			5, 10
			6, 01

(a) Table containing only stable states

State, GR			
x_1x_2 00	01	11	10
1, 00	2	—	5
1	**2**, 10	4	—
1	**3**, 01	4	—
—	3	**4**, 11	6
1	—	4	**5**, 10
1	—	4	**6**, 01

(b) Completed primitive flow table

Table 11.9 Reduced flow table

State, GR			
x_1x_2 00	01	11	10
1, 00	**2**, 10	4, 11	**5**, 10
1, 01	**3**, 01	**4**, 11	**6**, 01

Table 11.10 Excitation and output table

	Y, GR			
	x_1x_2			
y	00	01	11	10
0	0, 00	0, 10	1, 11	0, 10
1	0, 01	1, 01	1, 11	1, 01

direct the transitions to states **3** and **6**, which correspond to the output condition $GR = 01$.

The merger graph for the primitive flow table is shown in Fig. 11.11. It contains two triangles leading to the closed covering {(125), (346)}. The reduced flow table, which consists of two rows, is given in Table 11.9. The optional output symbols associated with the unstable states have been specified in such a way that R will be fast in turning on and slow in turning off.

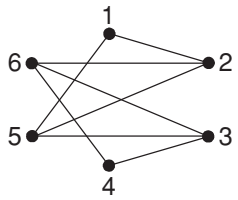


Fig. 11.11 Merger graph for the flow table of Table 11.8*b*.

The assignment of $y = 0$ to the first row and $y = 1$ to the second row of the reduced flow table leads to the excitation and output tables of Table 11.10. The excitation and output functions are

$$Y = (x_1 + x_2)y + x_1x_2,$$
$$G = (x_1 + x_2)y' + x_1x_2,$$
$$R = y + x_1x_2.$$

Races and cycles

In Section 11.1, we discussed the difficulties that may arise as a result of the different delays associated with the various gates if multiple input changes are allowed. The same difficulties may arise if two or more secondary variables are required to change their values simultaneously. For practical reasons, it

Table 11.11 Illustration of races and cycles

	Y_1Y_2			
	x_1x_2			
y_1y_2	00	01	11	10
00	11	**00**	10	01
01	11	00	11	**01**
11	**11**	00	10	**11**
10	11	**10**	**10**	11

is clearly impossible to guarantee that all secondary elements indeed have precisely the same delays. As a result, the assignment of secondary variables to the rows of a reduced flow table must be such that the circuit will operate correctly even if different delays are associated with the various secondary elements.

A reduced excitation table is shown in Table 11.11. When both input values are 0 and $y_1y_2 = 00$, the required transition to the state $y_1y_2 = 11$ involves a change in the values of two secondary variables. If these two changes occur simultaneously, the transition specified in the table will actually take place. However, if either y_1 or y_2 changes first then, instead of going directly to the secondary state 11, the circuit will go to either state 01 or state 10. Fortunately, since in either case the required transition is to state 11, as indicated by the entries 11 in rows 01 and 10, column 00, the circuit will finally reach its destination. Such a situation, where a change in more than one secondary variable is required, is called a *race*. If the final state reached by the circuit does not depend on the order in which the variables change, as is the case discussed above, then the race is said to be a *noncritical race*.

Now suppose that the circuit is in the state $y_1y_2 = 11$ and that the input state is $x_1x_2 = 01$. The required transition is to the state $y_1y_2 = 00$. If y_1 changes faster than y_2 then the circuit will go to state 01, from which it will reach state **00**, as indicated by entry 00 in row 01, column 01. However, if y_2 changes faster than y_1 then the circuit will go to the state $y_1y_2 = \mathbf{10}$ and remain there, since the total state $x_1x_2 = 01$, $y_1y_2 = 10$ is a stable state. Thus, the circuit operation will be incorrect. Such a situation, where the final stable state reached by the circuit depends on the order in which the internal variables change, is referred to as a *critical race* and must always be avoided.

Races can sometimes be avoided by directing the circuit through intermediate unstable states, before it reaches its final destination. When the circuit of Table 11.11 is in the secondary state $y_1y_2 = 01$ and the input state $x_1x_2 = 11$, the required transition is to state **10**. However, since such a transition, from 01 to 10, involves two simultaneous changes in the ys, the unstable state 11 is entered in row 01, column 11, thereby directing the circuit to row 11, from

Table 11.12 A valid assignment for the flow table of Table 11.11

	Y_1Y_2			
y_1y_2	x_1x_2 00	01	11	10
00	10	**00**	10	01
01	10	00	11	**01**
10	**10**	00	11	**10**
11	10	**11**	**11**	10

which it is directed to go to **10**. Such a situation, where a circuit goes through a *unique* sequence of unstable states, is called a *cycle*. When a state assignment is made such that it introduces cycles, care must be taken to ensure that each cycle terminates on a stable state. If a cycle does not contain a stable state then the circuit will go from one unstable state to another, until the inputs are changed. Obviously, such a situation must always be avoided when designing asynchronous circuits.

To eliminate the critical race in column 01, it is necessary to select another secondary assignment such that all critical transitions involve single variable changes. This can be accomplished by the assignment shown in Table 11.12. It is, of course, necessary to check that no new critical races have been introduced by this assignment. Having verified this, we can proceed to realize the flow table.

An assignment that contains no critical races or undesired cycles is referred to as a *valid* assignment. As we shall subsequently see, in many situations a valid assignment cannot be obtained merely by interchanging the assignments of several states in an invalid assignment; more sophisticated methods must be used.

Methods of secondary assignment

We now propose methods for obtaining secondary-state assignments such that each transition is accomplished either by a change of secondary state in which only one secondary variable changes or by a change of secondary state in which a multiple change of secondary variables does not result in a critical race. One way of arriving at the desired result is to test each transition and to ensure that the assignment of rows containing a lightface entry i will be adjacent to the assignment of the row containing the boldface entry $\boldsymbol{i}$. Subsequently, we shall refer to states that differ in only one variable as *adjacent* states.

The flow table of Table 11.13 contains three rows, denoted a, b, and c. Inspection of column 00 in the table reveals that the assignment of row a must be adjacent to that of row b, such that the transition from unstable state 1 to the

Table 11.13 A flow table

	State			
	x_1x_2			
PS	00	01	11	10
a	**1**	3	**4**	**6**
b	1	3	**5**	**7**
c	**2**	**3**	5	6

stable state **1** will involve just a single variable change. In a similar fashion, we arrive at the following required adjacencies for race-free operation:[3]

column 00 : row *b* must be adjacent to row *a*
column 01 : rows *a* and *b* must be adjacent to row *c*
column 11 : row *c* must be adjacent to row *b*
column 10 : row *c* must be adjacent to row *a*

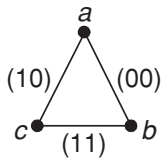


Fig. 11.12 Transition diagram for the flow table of Table 11.13.

These required adjacencies can be demonstrated by the diagram shown in Fig. 11.12, where each row is represented by a vertex and, for each pair of adjacent rows, an arc is drawn between the corresponding vertices. The arc labels (in parentheses) indicate the columns of the flow table in which the transitions are required. Such a diagram is known as a *transition diagram*. The problem now is to assign secondary states to the vertices of the transition diagram, such that each pair of adjacent vertices is assigned a pair of adjacent secondary states.

If row *a* of Table 11.13 is assigned a combination of values of state variables with an even number of 1's, say 00, row *b* must contain an odd number of 1's, say 01. Now, for row *c* to be adjacent to both rows *a* and *b*, it must contain an odd number of 1's *and* an even number of 1's, which obviously cannot be achieved. To overcome this difficulty, it is necessary to augment the flow table either by assigning two secondary states to row *c* or by introducing cycles that lead the circuit to the desired stable states. These possibilities are illustrated in Tables 11.14*a*, *b*. In the first case, each transition to state ***c*** (see below) is directed to the adjacent one, as illustrated in column 01. In the second case, an entry in row 10 is used as an intermediate unstable state to direct the circuit to the desired stable state.

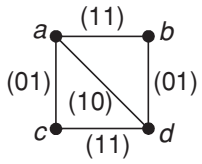


Fig. 11.13 Transition diagram for the flow table of Table 11.15.

Here, the use of a fourth row does not increase the number of secondary variables. In other situations, however, the augmentation of a flow table may involve such an increase. To examine this problem in terms of a specific situation, consider the flow table in Table 11.15 and its transition diagram shown in Fig. 11.13. We observe that row *a* must be adjacent to three other rows, as must row *d*. Clearly there is no way of assigning four secondary states such that the

[3] If noncritical races are permitted, as is usually the case, then column 01 requirement may be eliminated, since column 01 contains only one stable state.

Table 11.14 Augmented flow tables

		Y_1Y_2			
	y_1y_2	00	01	11	10
a	00	**00**	10	**00**	**00**
b	01	00	11	**01**	**01**
c	11	**11**	**11**	01	10
c	10	**10**	**10**	11	00

(a) Two assignments to row c

		Y_1Y_2			
	x_1x_2				
	y_1y_2	00	01	11	10
a	00	**00**	01	**00**	**00**
b	01	00	11	**01**	**01**
c	11	**11**	**11**	01	10
	10	–	–	–	00

(b) Utilizing an unspecified entry as an unstable state

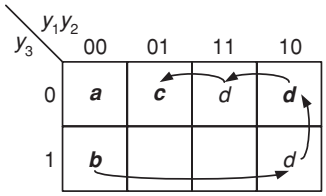


Fig. 11.14 Transition diagram.

Table 11.15 A flow table that requires three secondary variables

	State			
	x_1x_2			
PS	00	01	11	10
a	**1**	**2**	4	**6**
b	1	3	**4**	**7**
c	1	2	**5**	**8**
d	1	**3**	5	6

above adjacencies will be satisfied. Hence a third secondary variable must be added.

The eight combinations of three secondary variables are represented by the cells of the map of Fig. 11.14. To find a valid assignment, we start by placing a bold ***a*** in cell $y_1y_2y_3 = 000$ to indicate that row a will be assigned the secondary state 000. Similarly, we place ***b***, ***c***, and ***d*** in the three cells adjacent to cell ***a***. This, however, means that each of the transitions from rows b to d and d to c requires two changes of secondary variables. These multiple changes can be accomplished by directing the circuit to its final destination through unstable states, as shown by the arrows in Fig. 11.14. The flow table resulting from this assignment is shown in Table 11.16.

11.4 Synthesis of burst-mode circuits

Since SIC fundamental-mode machines are quite restrictive, a straightforward generalization leads to *multiple-input-change (MIC) fundamental-mode* machines, in which several inputs can change values in a narrow time interval and no further inputs change values until the machine has stabilized. However, because of the narrow time interval allowed for all input value changes, MIC

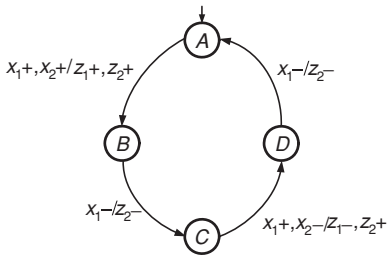


Fig. 11.15 A burst-mode specification.

Table 11.16 A race-free flow table

		State			
		x_1x_2			
	$y_1y_2y_3$	00	01	11	10
a	000	**1**	**2**	4	**6**
b	001	1	3	**4**	**7**
	011				
c	010	1	2	**5**	**8**
	110			5	
	111				
	101		3		
d	100	1	**3**	5	6

fundamental-mode machines are still quite restrictive. A further generalization of such machines is *burst-mode machines*. Such machines also allow several inputs to change values concurrently. However, all the changes need not occur in a narrow time interval. They can change in any order at any time within a given *input burst* and respond with a set of output value changes called the *output burst*. This eases the timing constraints imposed on the environment in which the machine is placed.

Burst-mode specification

A burst-mode specification with two inputs, x_1 and x_2, and two outputs, z_1 and z_2, is shown in Fig. 11.15. The start state is A, as indicated. The initial values of the inputs and outputs can be specified or assumed to have a default value 0. A label is associated with each arc consisting of an input burst and an output burst separated by /. A rising (falling) transition is denoted by $+$ ($-$).

The machine is initially stable in any given state. The rising or falling transitions associated with an input burst of an outgoing arc can arrive in any order and at any time. However, their change is monotonic. When the last input transition arrives, the burst is deemed complete. The machine then generates the corresponding output burst, if any, and moves to the specified next state. After the machine stabilizes, this process can begin anew.

There are three restrictions that a burst-mode specification must obey.

- *Nonempty input bursts* If no input undergoes a transition, the machine remains in its current state.
- *Maximal set property* No input burst on an outgoing arc from any state must be a subset of an input burst on another outgoing arc from the same state. Note that if such a subset were allowed, the machine would not know whether it should wait for another input transition.

Table 11.17 A flow table

	State, z_1z_2			
	x_1x_2			
PS	00	01	11	10
A	A, 00	A, 00	B, 11	A, 00
B	—	C, 10	B, 11	—
C	C, 10	C, 10	C, 10	D, 01
D	A, 00	—	—	D, 01

- *Unique entry point* Each state should have a unique set of input and output values through which it is entered. For example, in the specification shown in Fig. 11.15, let us assume that in starting state A, $x_1x_2 = 00$ and $z_1z_2 = 00$. Then we can check that the input/output values for states B, C, and D are $11/11$, $01/10$, $10/01$, respectively. The arc from D to A takes these values back to $00/00$, which is the unique entry point for A.

Flow table

In order to synthesize a circuit from a burst-mode specification, first it has to be translated into a flow table. For the specification shown in Fig. 11.15, the flow table is shown in Table 11.17. Each state in the specification is represented by a row in the flow table and each input combination by a column. Each entry in the table represents the *complete state* of the machine, which includes the state the machine goes to and the corresponding output values. Consider initial state A, which is mapped to row A where the complete state $A, 00$ is stable. The input burst x_1+, x_2+ on the outgoing arc from state A is also mapped to this row. This input burst leads to four possible temporary input combinations: no change, x_1+, x_2+, and (x_1+, x_2+). The complete state remains the same until the input burst is complete, after which the state is specified as are the output values based on the output burst z_1+, z_2+, thus leading to the complete state $B, 11$. On the outgoing arc from state B to C the input burst is simply x_1-. Thus, there are only two temporary input combinations in this case: no change and x_1-. The latter yields the entry $C, 10$ in this row. This complete state incorporates the effect of the output burst z_2-. The remaining two entries in this row cannot be reached and are hence left unspecified. A similar analysis applies to the other rows.

Flow table reduction and state assignment

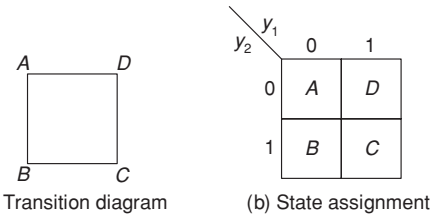
The flow table for a burst-mode specification has no function hazards; this stems from the requirement that the complete state must not change until the full input burst has arrived. Also, it is always possible to obtain a hazard-free

sum-of-products realization H for each secondary variable and output. This follows from the fact that, for each such variable, the required cube can be included in some product of H and no product of H illegally intersects any privileged cube. The latter is true because all transitions in any row of the flow table have the same complete start state, which will be included in the required cubes for these transitions.

It is possible to minimize the number of states in a flow table through state merging. However, even when two states are compatible it may sometimes be incorrect to merge them since it may no longer be possible to guarantee a hazard-free realization of all secondary and output variables. The conditions under which state merging is possible are given in [13]. However, for the rest of the discussion, we will assume that no state merging is done.

Various methods are available for obtaining a critical race-free secondary state assignment for the flow table. One way is use the transition diagrams discussed earlier.

Example Consider the burst-mode specification in Fig. 11.15. Its transition diagram and a possible state assignment are shown in Fig. 11.16.



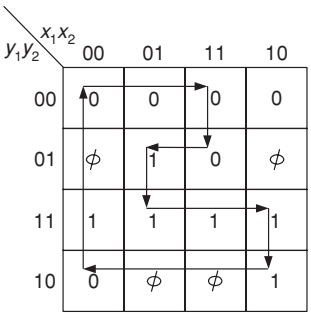
(*a*) Transition diagram (b) State assignment

Fig. 11.16 A critical race-free state assignment.

A synthesis example

The excitation and output table is the starting point for further synthesis. As discussed earlier, we need to identify next the required cubes and dhf-prime implicants for each next-state and output variable and obtain the minimal sum-of-products expressions based on the subset of the dhf-prime implicants that covers all the required cubes.

Continuing with the state assignment in Fig. 11.16, consider its excitation and output table, shown in Table 11.18. For Y_1, Y_2, z_1, and z_2, the maps with the relevant transitions as well as the dhf-prime implicant charts are shown in Fig. 11.17. The horizontal transitions shown in the maps correspond to the input burst and the vertical transitions to the change in state. For example, the input burst x_1+, x_2+ in the specification shown in Fig. 11.15 takes the machine from

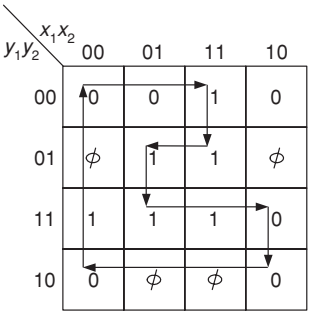


Y_1 map

dhf-prime implicants	Required cubes		
	$x_1'x_2y_2$	y_1y_2	$x_1x_2'y_1$
$x_1'y_2$	×		
$x_2'y_2$			
y_1y_2		×	
x_1y_1			×
x_2y_1			

Minimal hazard-free sum-of-products

$Y_1 = x_1'y_2 + y_1y_2 + x_1y_1$

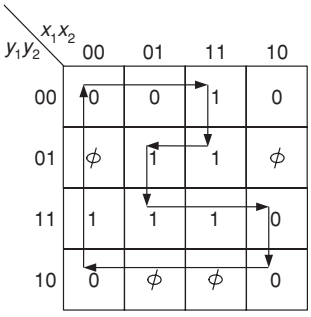


Y_2 map

dhf-prime implicants	Required cubes				
	$x_1x_2y_1'$	$x_2y_1'y_2$	$x_1'x_2y_2$	$x_2y_1y_2$	$x_1'y_1y_2$
$x_1x_2y_1'$	×				
$y_1'y_2$		×			
$x_1'y_2$			×		×
x_2y_2		×	×	×	
x_2y_1				×	

Minimal hazard-free sum-of-products

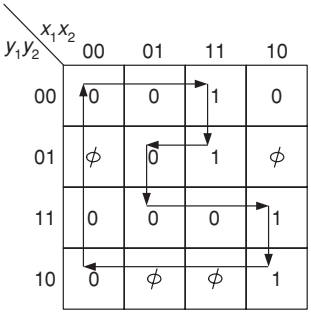
$Y_2 = x_1x_2y_1' + x_1'y_2 + x_2y_2$



z_1 map

Minimal hazard-free sum-of-products

$z_1 = Y_2 = x_1x_2y_1' + x_1'y_2 + x_2y_2$



z_2 map

dhf-prime implicants	Required cubes	
	$x_1x_2y_1'$	$x_1x_2'y_1$
$x_1x_2y_1'$	×	
$x_1y_1'y_2$		
$x_1x_2'y_1$		×
$x_1y_1y_2'$		
$x_1x_2y_2'$		

Minimal hazard-free sum-of-products

$z_2 = x_1x_2y_1' + x_1x_2'y_1$

Fig. 11.17 Synthesis from a burst-mode specification.

Table 11.18 Excitation and output table

	Y_1Y_2, z_1z_2			
	x_1x_2			
y_1y_2	00	01	11	10
00	00, 00	00, 00	01, 11	00, 00
01	—	11, 10	01, 11	—
11	11, 10	11, 10	11, 10	10, 01
10	00, 00	—	—	10, 01

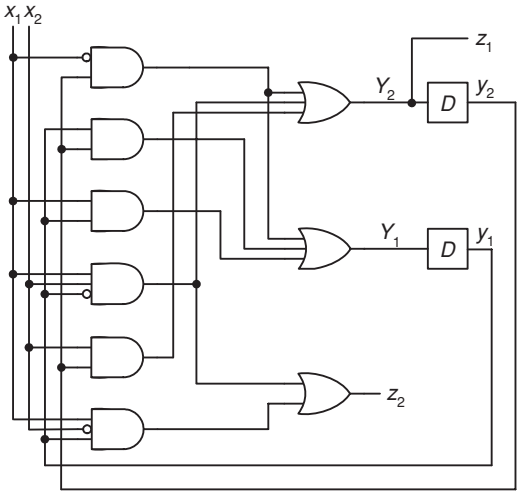


Fig. 11.18 Synthesized circuit.

state A to B. This corresponds to a horizontal transition from $(x_1, x_2, y_1, y_2) = 0000$ to 1100, followed by a vertical transition from 1100 to 1101 (note that A's assignment is 00 whereas B's assignment is 01). Some dhf-prime implicants are not needed for any required cube, with the result that the corresponding row is blank in the dhf-prime implicant chart. The minimal hazard-free sum-of-products expressions are also shown in Fig. 11.17. The corresponding circuit is shown in Fig. 11.18.

Notes and references

The first systematic treatment of asynchronous sequential circuits was due to Huffman [7], whose model for fundamental-mode circuits was presented in this chapter. McCluskey [10] also studied fundamental-mode circuits. Huffman [6] and McCluskey [9] were also the main initial contributors to hazard analysis and hazard-free circuit

design. Eichelberger [4] dealt with MIC logic hazards. Beister [1] showed how to get rid of MIC dynamic logic hazards. Nowick and Dill [13] presented an exact two-level minimization algorithm for obtaining hazard-free circuits. Unger [15], Bredeson [2], and Kung [8] presented hazard-nonincreasing logic transformations. Huffman [5] studied the secondary-assignment problem for asynchronous circuits and proposed several race-free universal assignments. Unger [14] pointed out the existence of inherent hazards within fundamental-mode circuits and showed how to eliminate such hazards by inserting delays. Good presentations of asynchronous circuits are available in Miller [11] and Unger [15]. The survey article by Davis and Nowick [3] and the book by Myers [12] provide excellent further reading material for interested readers.

[1] Beister, J.: "A unified approach to combinational hazards," *IEEE Trans. Computers*, vol. C-23, no. 6, pp. 566–575, June 1974.

[2] Bredeson, J. G.: "Synthesis of multiple-input change hazard-free combinational switching circuits without feedback," *Int. J. Electronics (GB)*, vol. 39, no. 6, pp. 615–624, December 1975.

[3] Davis, A., and S. M. Nowick: "An introduction to asynchronous circuit design," University of Utah Technical Report, Department of Computer Science, UUCS-97-013, September 1997.

[4] Eichelberger, E. B.: "Hazard detection in combinational and sequential switching circuits," *IBM J. Research & Development*, vol. 9, pp. 90–99, 1965.

[5] Huffman, D. A.: "A study of the memory requirements of sequential switching circuits," *MIT Res. Lab. Electron. Technical Report 293*, April 1955.

[6] Huffman, D. A.: "The design and use of hazard-free switching networks," *J. Assoc. Computing Machinery*, vol. 4, pp. 47–62, January 1957.

[7] Huffman, D. A.: "The synthesis of sequential switching circuits," *J. Franklin Inst.*, vol. 257, pp. 275–303, March-April 1954.

[8] Kung, D. S.: "Hazard-nonincreasing gate-level optimization algorithms," in *Proc. In. Conf. Computer-Aided Design*, pp. 631–634, November 1992.

[9] McCluskey, E. J.: "Transient in combinational logic circuits," in *Redundancy Techniques for Computing Systems*, pp. 9–46, Spartan, Washington, DC, 1962.

[10] McCluskey, E. J.: "Fundamental and pulse mode sequential circuits," in *Proc. IFIP Congress* 1962, North Holland, Amsterdam, 1963.

[11] Miller, R. E.: *Switching Theory*, vol. 2, John Wiley & Sons, New York, 1965.

[12] Myers, C. J.: *Asynchronous Circuit Design*, John Wiley & Sons, New York, July 2001.

[13] Nowick, S. M., and D. L. Dill: "Exact two-level minimization of hazard-free logic with multiple-input changes," *IEEE Trans. Computer-Aided Design*, vol. 14, no. 8, pp. 986–997, August 1995.

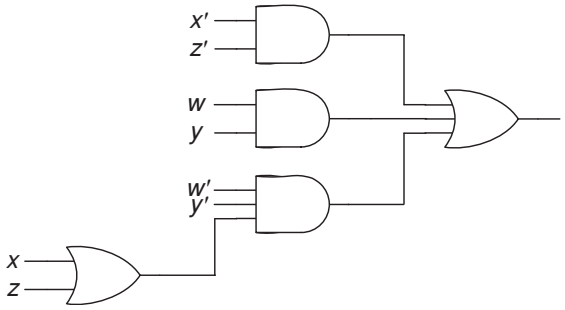
[14] Unger, S. H.: "Hazards and delays in asynchronous sequential switching circuits," *IRE Trans. Circuit Theory*, vol. CT-6, no. 12, 1959.

[15] Unger, S. H.: *Asynchronous Sequential Switching Circuits,* John Wiley & Sons, New York, 1969.

Problems

Problem 11.1. Analyze the circuit in Fig. P11.1 for SIC static hazards. Redesign it to make it SIC hazard-free.

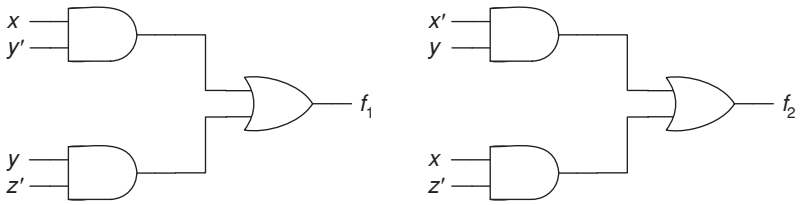
Fig. P11.1



Problem 11.2. Consider the two-output circuit shown in Fig. P11.2. Without inserting any extra gates in it, make both outputs SIC hazard-free.

Hint: You are allowed to add connections to the circuit.

Fig. P11.2



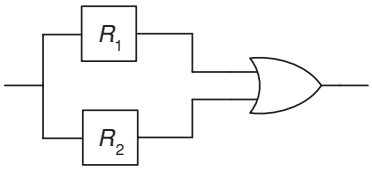
Problem 11.3

(a) If two AND–OR two-level circuits are SIC hazard-free, is the single-output circuit obtained by performing an OR of the two outputs guaranteed to be SIC hazard-free? Either prove this or provide a counter-example.

(b) Conversely, if two AND–OR two-level circuits each have an SIC hazard, is the single-output circuit obtained by, performing an OR of the two outputs guaranteed to have an SIC hazard? Either prove this or provide a counter-example.

Problem 11.4. Two different realizations, R_1 and R_2, of a function F are fed to an OR gate, as shown in Fig. P11.4. If both R_1 and R_2 are SIC hazard-free, is the overall circuit guaranteed to be SIC hazard-free? Explain your reasoning.

Fig. P11.4

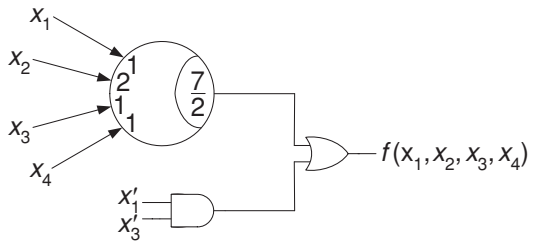


Problem 11.5

(a) Find all SIC static hazards in the circuit shown in Fig. P11.5. (Assume the individual elements to be hazard-free.)

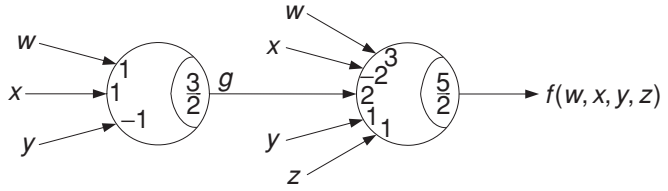
(b) Changing *only* the parameters of the threshold element, redesign the circuit in such a way that all SIC static hazards are eliminated.

Fig. P11.5



Problem 11.6. For the network shown in Fig. P11.6:
(a) show a map for $f(w, x, y, z)$;
(b) find all SIC hazards of the network;
(c) realize f with a single threshold element.

Fig. P11.6



Problem 11.7. In the function $f(x, y, z) = \sum(1, 3, 4, 5, 6, 7)$:
(a) find all MIC transitions that have a function hazard;
(b) find the required cubes for the MIC transition $111 \rightarrow 010$. What is the privileged cube for this transition?
(c) find the required cubes for MIC transition $111 \rightarrow 000$.

Problem 11.8. For the function $f(w, x, y, z) = \sum(0, 1, 2, 3, 5, 6, 7, 8, 9, 12, 13, 15)$ and the transitions $0001 \rightarrow 0100$, $0110 \rightarrow 0011$, $1101 \rightarrow 1010$, and $1011 \rightarrow 1010$:
(a) find all the dhf-prime implicants;
(b) find a hazard-free sum-of-products expression.

Problem 11.9. From the excitation and output tables, in Table P11.9, for an SIC fundamental-mode asynchronous sequential circuit, determine which input sequences result in a 1 output value.

Table P11.9

	Y_1Y_2, z			
	x_1x_2			
y_1y_2	00	01	11	10
00	**00**, 0	10, 0	01, 0	**00**, 0
01	00, 0	11, 0	**01**, 1	11, 0
11	00, 0	**11**, 0	10, 0	**11**, 0
10	00, 0	**10**, 0	**10**, 0	11, 0

Problem 11.10. Each of the following specifications describes an SIC fundamental-mode sequential circuit with two inputs, x_1 and x_2, and one output, z. Show a primitive and a reduced flow table for each circuit.

(a) The output $z = 1$ if both x_1 and x_2 are at 1 and the value of x_1 becomes 1 before that of x_2.

(b) When $x_2 = 1$, the value of the output z is equal to the value of x_1; when $x_2 = 0$, the output remains fixed at its last value prior to when the value of x_2 became 0.

(c) The value of the output z is equal to 0 whenever $x_1 = 0$. The first change in the value of input x_2 occurring while $x_1 = 1$ causes the value of z to become 1. Thereafter, the value of z remains at 1 until the value of x_1 returns to 0.

Problem 11.11. Give a minimum-row reduced-flow-table description of an SIC fundamental-mode two-input (x_1, x_2), one-output (z) sequential circuit that operates in the following manner: the output $z = 1$ if and only if the input state $x_1 = x_2 = 1$ and the next-to-last input variable change was a change in the value of x_1. Assume that the circuit is initially in the input state $x_1 = x_2 = 0$. Is the reduced flow table unique?

Problem 11.12. The value of the output z of an SIC fundamental-mode two-input sequential circuit is to change from 0 to 1 only when the value of x_2 changes from 0 to 1 while $x_1 = 1$. The output value is to change from 1 to 0 only when the value of x_1 changes from 1 to 0 while $x_2 = 1$.

(a) Find a minimum-row reduced flow table. The output should be fast and flicker-free.

(b) Show a valid assignment and write a set of (*static*) *hazard-free* excitation and output equations.

Problem 11.13. An SIC fundamental-mode sequential circuit with two inputs, x_1 and x_2, and two outputs, z_1 and z_2, is to be designed so that z_i (for $i = 1, 2$) takes on the value 1 if and only if x_i was the input whose value changed last.

(a) Find a minimum-row reduced flow table and a valid assignment.

(b) Assuming that all inputs are available in an uncomplemented as well as a complemented form, show a realization using NAND gates. (fourteen gates are sufficient.)

Problem 11.14. Design an SIC fundamental-mode asynchronous sequential circuit with two inputs, x_1 and x_2, and two outputs, G and R, which is to operate in the following manner. Initially, both input values and both output values are equal to 0. The first input to assume the value 1, either x_1 or x_2, turns G "on" (i.e., sets G to 1). With the first input value equal to 1, if the second input value becomes equal to 1 then R turns on. Thereafter, as long as either input value remains equal to 1, the input that first caused G to turn on controls the operation of G, i.e., it causes G to turn off when it assumes the value 0 and to turn on again when it assumes the value 1. The second input controls the operation of R in the same manner.

(a) Show a minimum-row reduced flow table and find a valid assignment.

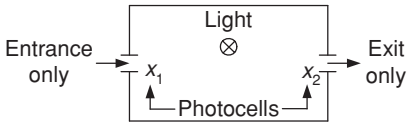
(b) Find the excitation and output equations.

Problem 11.15. At a junction of a single-track railroad and a road, traffic lights are to be installed. The lights are to be controlled by switches that are pressed or released by the trains. When a train approaches the junction from either direction and is within 1500 feet from it, the lights are to change from green to red and remain red until the train is 1500 feet past the junction.

(a) Write a primitive flow table and reduce it. You may assume that the length of a train is smaller than 3000 feet.
(b) Show a circuit realization of the light-control network.
(c) Repeat the design if it is known that the trains may be longer than 3000 feet.

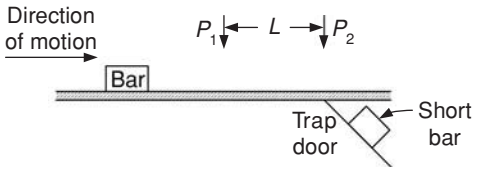
Problem 11.16. Figure P11.16 illustrates an office for two students. Instead of light switches the room has two photocells, one at each door. If either or both students are in the office, the light is to be on. The students can enter or exit only as shown; entrances and exits never occur simultaneously. The photocells indicate a 1 when their beam is interrupted by a student entering or exiting and a 0 at all other times.
(a) Find a primitive and a minimum-row reduced flow table that describe the light-control operation.
(b) Show a valid assignment and find the excitation and output equations.
(c) Repeat (a) if entering and exiting the room simultaneously is allowed.

Fig. P11.16



Problem 11.17. A factory produces steel bars of length $L + \delta$ and $L - \delta$. It is required that the bars are to be sorted by placing them on a conveyor belt passing under two photocells, as shown in Fig. P11.17. The spacing between the bars on the belt is greater than δ. To the right of P_2 is a trap door through which short bars can drop. The trap door should not be open when the beam of P_2 is interrupted and should be open immediately after a short bar, of length $L - \delta$, has completely passed P_2. Let the value of output x_i of P_i be 1 when the beam of P_i is interrupted. Let the value of the trap-door control z be 1 when the door is open.
(a) Find a minimum-row reduced flow table, with eight stable states, that describes the trap-door control operation.
(b) Show a valid assignment and find the logic equations for the memory elements and the trap-door control.

Fig. P11.17

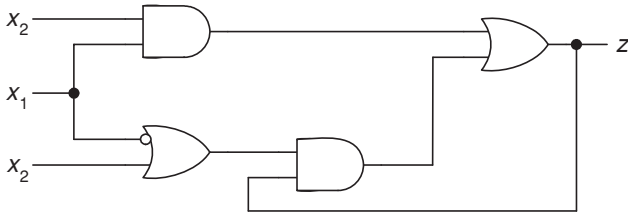


Problem 11.18. A completely automatic and independent traffic-light system for the intersection of roads x and y consists of two sensors, some processing circuitry, and the lights. The sensors and circuitry generate two outputs, z and w. Output z attains the value 1 if and only if $m(x) - m(y) \geq 6$, where $m(x)$ indicates the number of cars waiting to cross a road y. Output w attains the value 1 if and only if $m(y) - m(x) \geq 6$. We wish to design an SIC fundamental-mode sequential circuit with inputs (z, w, z', w') and outputs (G_x, R_x, G_y, R_y), where G and R refer to green and red lights, respectively, and the subscripts indicate the street from which the light is visible. The objective is to minimize intersection load by unloading whichever street is overloaded, i.e., has at

least six cars more than the other. The lights of the street being unloaded should remain green until the other street becomes overloaded.
(a) Show a primitive flow table.
(b) Give a reduced flow table.
(c) Show a circuit realization. The outputs are to be fast and flicker-free.

Problem 11.19. In the circuit of Fig. P11.19, the values of input variables x_1 and x_2 never change simultaneously.
(a) Describe in words the terminal behavior of the circuit.
(b) Derive the flow table for the circuit.
(c) Show how one of the gates can be eliminated without changing the flow table. What physical problems might this cause, and how can they be prevented?
Hint: To derive the flow table, open the feedback loop.

Fig. P11.19



Problem 11.20. The reduced flow table of Table P11.20*a* is to be assigned three secondary variables, as shown in Table P11.20*b*. Note that several combinations of $y_1y_2y_3$ values have been assigned to the first two rows of the reduced table. Consequently the circuit will be stable when $x_1x_2 = 00$ in any of the $y_1y_2y_3$ combinations 000, 001, 011, for example, and each of these stable configurations must be equivalent to **1**. Complete an excitation table for the situation when each transition takes as short a time as possible. Is the excitation table unique?

Table P11.20

	State			
	x_1x_2			
PS	00	01	11	10
a	**1**	5	**6**	9
b	1	**4**	7	**8**
c	**2**	**5**	**7**	9
d	**3**	4	6	**9**

(*a*) Reduced flow table

	$Y_1Y_2Y_3$				
		x_1x_2			
	$y_1y_2y_3$	00	01	11	10
a	000				
a	001				
a	011				
b	010				
b	100				
b	101				
c	111				
d	110				

(*b*) Excitation table

Problem 11.21
(a) Find all the races in the flow table of Table P11.21 and indicate those that are critical and those that are not.
(b) Find another assignment that contains no critical races.

Table P11.21

	State			
	x_1x_2			
y_1y_2	00	01	11	10
00	**00**	11	**00**	11
01	11	**01**	11	11
10	00	**10**	11	11
11	**11**	**11**	00	**11**

Problem 11.22. For each of the reduced flow tables in Table P11.22, find an assignment that contains no critical races and requires a minimum of secondary variables.

Table P11.22

State			
x_1x_2			
00	01	11	10
1	**3**	**5**	7
2	3	**6**	7
1	**4**	6	**7**

(*a*)

State			
x_1x_2			
00	01	11	10
1	3	6	**7**
1	**3**	5	7
2	**4**	**5**	7
2	4	**6**	7

(*b*)

State			
x_1x_2			
00	01	11	10
1	3	**5**	7
1	**4**	**6**	8
2	**3**	6	**7**
2	4	5	**8**

(*c*)

State			
x_1x_2			
00	01	11	10
1	5	**7**	**10**
2	**4**	**8**	10
3	5	**9**	12
2	**5**	9	11
3	4	7	**11**
1	**6**	8	**12**

(*d*)

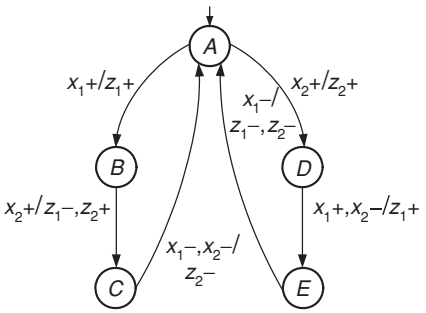
State			
x_1x_2			
00	01	11	10
1	4	**7**	**10**
1	5	8	**11**
2	6	**8**	10
2	**4**	**9**	11
3	**6**	9	10
3	**5**	7	11

(*e*)

Problem 11.23. Consider the burst-mode specification shown in Fig. P11.23.

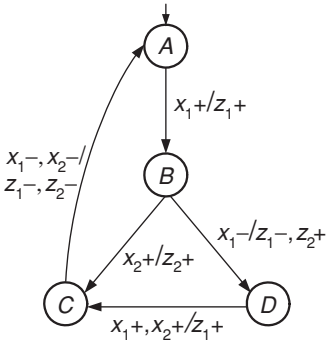
(a) Assuming that the unique entry point for state A is 00/00, what are the entry points for each of the other four states?
(b) Obtain a flow table from the specification.
(c) Find a secondary state assignment that is free of critical races.
(d) Obtain an excitation and output table based on the above state assignment.
(e) Synthesize a minimal two-level hazard-free circuit.

Fig. P11.23



Problem 11.24. Repeat Problem 11.23 for the burst-mode specification shown in Fig. P11.24.

Fig. P11.24



CHAPTER

12 Structure of sequential machines

One of the main problems in the synthesis of sequential machines is that of assigning combinations of state-variable values to the states of the machine. This assignment determines the complexity and structure of the circuit which realizes the machine. Various restrictions and requirements may be imposed on the state assignment, depending on the design objectives and intended use of the circuit. It may be desirable, for example, to construct it using a minimum amount of logic, or to build it from an interconnection of smaller circuits, and so on. The *structure* of a sequential machine includes the manner in which a machine can be realized from a set of smaller component machines as well as the functional dependencies of its state and output variables. It is our aim in this chapter to study the state-assignment problem and how it affects the structure and complexity of sequential machines.

12.1 Introductory example

The close relationship between the state-assignment problem and the structure of sequential machines will be demonstrated by means of the machine M_1 shown in Table 12.1. Two possible state assignments for M_1 are shown in Table 12.2. The logic equations corresponding to assignment α, which are derived from the excitation and output tables, are

$$\begin{aligned} Y_1 &= x'y_1 + xy_1' = f_1(x, y_1), \\ Y_2 &= x'y_1 + xy_2 = f_2(x, y_1, y_2), \\ z &= xy_2' = f_0(x, y_2). \end{aligned}$$

From these equations, it is evident that Y_1 is a function of y_1 and of the external input and is independent of y_2. However, Y_2 depends on the external input as well as y_1 and y_2. The output z is a function of x and y_2 only. The circuit diagram of M_1 is shown in Fig. 12.1*a*. The dependency of the next-state variables and the output is illustrated by the block diagram of Fig. 12.1*b*, where,

Table 12.1 Machine M_1

	NS		z	
PS	$x=0$	$x=1$	$x=0$	$x=1$
A	A	D	0	1
B	A	C	0	0
C	C	B	0	0
D	C	A	0	1

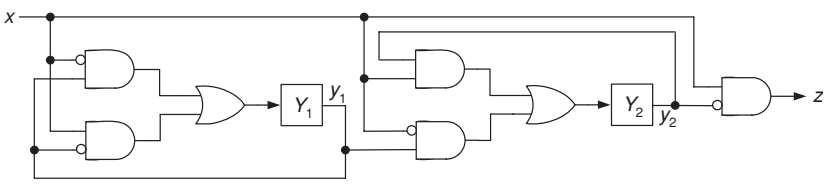
Table 12.2 Excitation and output tables for M_1

		Y_1Y_2		z	
	y_1y_2	$x=0$	$x=1$	$x=0$	$x=1$
A	00	00	10	0	1
B	01	00	11	0	0
C	11	11	01	0	0
D	10	11	00	0	1

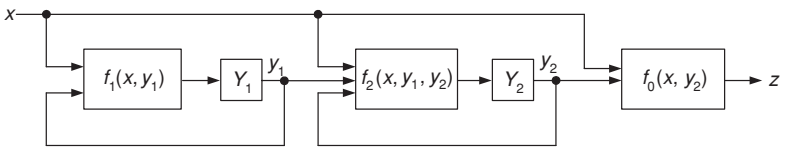
(a) Assignment α

		Y_1Y_2		z	
	y_1y_2	$x=0$	$x=1$	$x=0$	$x=1$
A	00	00	11	0	1
B	01	00	10	0	0
C	10	10	01	0	0
D	11	10	00	0	1

(b) Assignment β

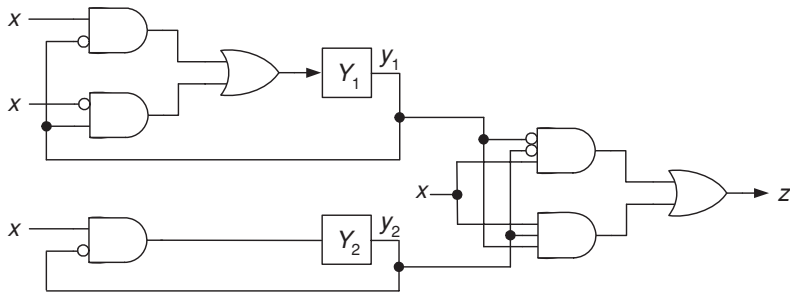


(a) Circuit diagram.

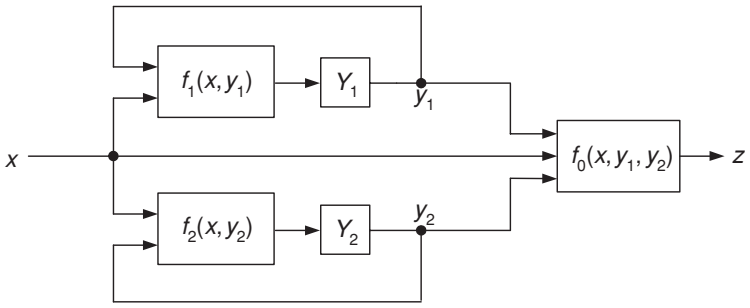


(b) Block diagram.

Fig. 12.1 First realization of M_1.



(*a*) Circuit diagram.



(*b*) Block diagram.

Fig. 12.2 Second realization of M_1.

for example, the block labeled $f_1(x, y_1)$ corresponds to the combinational logic associated with memory element Y_1, and so on.

The logic equations corresponding to assignment β, shown in Table 12.2*b*, are

$$\begin{aligned} Y_1 &= x'y_1 + xy_1' = f_1(x, y_1), \\ Y_2 &= xy_2' = f_2(x, y_2), \\ z &= xy_1'y_2' + xy_1y_2 = f_0(x, y_1, y_2). \end{aligned}$$

In this case Y_1 is independent of y_2 and Y_2 is independent of y_1. In other words, the next value of each state variable can be computed from its present value and the value of the present input, regardless of the value of the other state variable. The dependency of the output function, however, has increased in comparison with its dependency in assignment α, shown in Table 12.2*a*. The circuit and block diagrams corresponding to assignment β are shown in Fig. 12.2.

The preceding two realizations of machine M_1 clearly demonstrate that the choice of assignment affects the complexity of the circuit and determines the dependency of the next-state variables and the overall structure of the machine. Our objective in this chapter is to investigate the relationship of the

state assignment and the reduction in dependency of the state variables to the structure of a sequential machine. These factors will be shown to affect the complexity and cost of the final circuits as well.

12.2 State assignments using partitions

In this section we shall derive necessary and sufficient conditions for a sequential machine M to have assignments that result in reduced dependencies among the state variables. Such assignments generally yield simpler logic equations and circuits; they are also the fundamental means by which machine decompositions are obtained.

Closed partitions

Let machine M have a set of n states $S = \{S_1, S_2, \ldots, S_n\}$ and a set of p input symbols $I = \{I_1, I_2, \ldots, I_p\}$; then $k = \lceil \log_2 n \rceil$ state variables and $l = \lceil \log_2 p \rceil$ input variables are needed for a complete assignment, where $\lceil g \rceil$ is defined as the smallest integer equal to or greater than g. Each of the k next-state variables depends, in general, on the external inputs $x_1, x_2, \ldots, x_l$ and the k state variables, i.e.,

$$Y_i = f_i(y_1, y_2, \ldots, y_k, x_1, x_2, \ldots, x_l), \qquad i = 1, 2, \ldots, k.$$

Our objective is to obtain assignments in which the values of one or more subsets of the next-state variables can be determined independently of the values of the remaining variables, that is, assignments which yield logic equations for the variables $Y_1, Y_2, \ldots, Y_r$, where $1 \leq r < k$, that are independent of the remaining $k - r$ variables. Thus,

$$Y_i = f_i(y_1, y_2, \ldots, y_r, x_1, x_2, \ldots, x_l), \qquad i = 1, 2, \ldots, r.$$

The subset $\{Y_1, Y_2, \ldots, Y_r\}$ of state variables, whose values are independent of the values of $y_{r+1}, y_{r+2}, \ldots, y_k$, is said to be a *self-dependent* subset, and an assignment that yields such a subset is said to possess self-dependent subsets. Assignments α and β of machine M_1 both have this property.

The state-assignment problem may be viewed as either a coding problem or a partitioning problem. In viewing the state assignment as a coding problem, a distinct code is assigned to each row (state) of the state table. From the partitioning point of view, which we shall adopt in this chapter, each state variable y_i induces a partition τ_i on the set of states of the machine, such that two states are in the same block of τ_i if and only if they are assigned the same value of y_i. For example, in assignment α for machine M_1, $y_1 = 0$ for states A and B and $y_1 = 1$ for states C and D. Hence y_1 induces the partition $\tau_1 = \{\overline{A, B}; \overline{C, D}\}$ (see Definition 2.1) on the states of M_1. Similarly, y_2 induces the partition $\tau_2 = \{\overline{A, D}; \overline{B, C}\}$. Clearly, if the assignment is such that each

state has a unique code then the product of the k partitions $\tau_1, \tau_2, \ldots, \tau_k$ corresponding to $y_1, y_2, \ldots, y_k$ is equal to zero, that is,

$$\tau_1 \cdot \tau_2 \cdot \quad \cdots \quad \cdot \tau_k = \pi(0).$$

We have shown how an assignment induces a set of partitions whose product is the zero partition $\pi(0)$. The inverse process, that of assigning the values of the state variables to distinguish the blocks of a set of partitions, is the process of significance in the synthesis procedure. Given a partition τ with $\#(\tau)$ blocks on the set of states of M, to distinguish between these blocks it is necessary to select $r = \lceil \log_2 \#(\tau) \rceil$ state variables and assign a distinct combination of these variables to each block of τ; that is, all the states in each block are assigned the same values of $y_1, y_2, \ldots, y_r$. Each partition on the states of M provides some information regarding M's state. If M possesses two partitions τ_1 and τ_2 such that $\tau_1 > \tau_2$ then τ_2 provides more information than τ_1. Clearly, the zero partition provides all the necessary information, since knowledge of which block of $\pi(0)$ the machine is in is sufficient to determine the state of M uniquely. Thus, to obtain an assignment for M such that each state has a distinct code, it is necessary to assign the values of the state variables in such a way that they distinguish between the blocks of a set of partitions whose product is the zero partition.

Example For machine M_1, the product of the partitions $\tau_1 = \{\overline{A, B}; \overline{C, D}\}$ and $\tau_2 = \{\overline{A, C}; \overline{B, D}\}$ is zero, i.e., $\tau_1 \cdot \tau_2 = \pi(0)$. Hence, if we assign y_1 in such a way as to distinguish block (A, B) from block (C, D), and y_2 in such a way as to distinguish the blocks of τ_2, then each state of M_1 will have a distinct code. One such assignment is β, shown in Table 12.2*b*.

Definition 12.1 A partition π on the set of states of a sequential machine M is said to be *closed* if, for every two states S_i and S_j which are in the same block of π and any input symbol I_k in I, the states $I_k S_i$ and $I_k S_j$ are in a common block of π; $I_k S_i$ denotes the I_k-successor of S_i.

Example For machine M_1, Table 12.1, the partitions $\pi_1 = \{\overline{A, B}; \overline{C, D}\}$ and $\pi_2 = \{\overline{A, C}; \overline{B, D}\}$ are closed.[1] The 0- and 1-successors of (A, C) are (A, C) and (B, D), respectively, while the only successor of (B, D) is (A, C). If we denote the blocks of π_2, (A, C) and (B, D), by P and Q respectively then we may describe the successor relationships of these blocks by means of the graph of Fig. 12.3. Clearly, knowledge of the present block of M_1 and the input symbol is sufficient to determine the next block

[1] In general, we shall reserve π to denote closed partitions while τ, θ, etc., will denote arbitrary partitions.

uniquely. (We shall subsequently say that a machine is in a block when we mean that it is in one of the states contained in the block.)

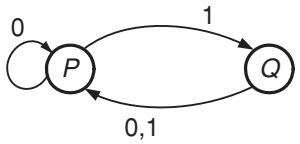


Fig. 12.3 Successor relationships of the blocks of the partition $\pi_2 = \{\overline{A, C}; \overline{B, D}\} = \{\overline{P}; \overline{Q}\}$.

Reduction of the functional dependency of the state variables

We shall now establish the relationship between closed partitions and the reduction in functional dependency of state variables.

Theorem 12.1 *Let M be a sequential machine with k state variables, $y_1, y_2, \ldots, y_k$. If there exists a closed partition π on the states of M and if r state variables, where $r = \lceil \log_2 \#(\pi) \rceil$, are assigned to the blocks of π, such that all the states contained in each block are assigned the same values of $y_1, y_2, \ldots, y_r$, then the next-state variables $Y_1, Y_2, \ldots, Y_r$ are independent of the remaining $k - r$ variables. Conversely, if the first r next-state variables, $Y_1, Y_2, \ldots, Y_r$ $(1 \leq r < k)$, can be determined from the values of the inputs and the first r state variables, independently of the values of the remaining $k - r$ variables, then there exists a closed partition π on the states of M such that two states, S_i and S_j, are in the same block of π if and only if they are assigned the same values of the first r variables.*

Proof Since each block of π is assigned the same values of the variables $y_1, y_2, \ldots, y_r$, and since π is closed, knowledge of the present block of π and the present input values is sufficient to determine the next block of π. In other words, knowledge of the present values of $y_1, y_2, \ldots, y_r$ and of the present input values is sufficient to determine the values of $Y_1, Y_2, \ldots, Y_r$, regardless of the values of the remaining variables. To prove the converse, form a partition π on the states of M such that all the states with the same assigned values of $y_1, y_2, \ldots, y_r$ are in the same block of π. To prove that π is closed, consider two states S_i and S_j that belong to the same block of π. Each of these states has the same assigned values of the first r variables and, since these variables are independent of the values of the remaining ones, an application of the same input sequence to both S_i and S_j causes the same change in the values of the first r variables for these two states. Therefore, for each value of I_k, the successors $I_k S_i$ and $I_k S_j$ have the same assignment of values for the first r variables and, consequently, are contained in the same block of π. Thus, π is closed. ◇

Example For machine M_1, the partitions $\pi_1 = \{\overline{A, B}; \overline{C, D}\}$ and $\pi_2 = \{\overline{A, C}; \overline{B, D}\}$ are closed. Since y_1 in assignment β has been assigned to distinguish the blocks of π_1, it is independent of y_2. Similarly, since y_2 has been assigned to distinguish the blocks of π_2, it is independent of y_1.

Theorem 12.1 actually states a necessary and sufficient condition for the decomposition of sequential machines. The existence of a partition τ and a closed partition π on the set of states of a machine M, such that $\pi \cdot \tau = \pi(0)$ guarantees that M can be composed of two component machines connected in *series*. The first component in the connection consists of $\lceil \log_2 \#(\pi) \rceil$ memory elements (and their excitation circuitry), corresponding to the state variables assigned to distinguish the blocks of π. Since these variables are independent of the remaining variables, the first component is often referred to as the *independent* component. The second component in the serial connection, also referred to as the *dependent* component, contains $\lceil \log_2 \#(\tau) \rceil$ memory elements, corresponding to the state variables assigned to distinguish the blocks of τ. We shall refer to the independent component as the *predecessor* machine and the dependent component as the *successor* machine. It is often convenient to view the predecessor machine as the component that distinguishes between the blocks of π, and the successor machine as the component that distinguishes between the states within the blocks of π.

The existence of two closed partitions on the states of M such that their product is zero, i.e., $\pi_1 \cdot \pi_2 = \pi(0)$, implies that M can be composed of two components operating in *parallel*, independently of each other. One component consists of $\lceil \log_2 \#(\pi_1) \rceil$ memory elements, corresponding to the variables assigned to distinguish the blocks of π_1. The second component consists of $\lceil \log_2 \#(\pi_2) \rceil$ memory elements, corresponding to the variables assigned to distinguish the blocks of π_2.

The preceding arguments can thus be summarized as follows.

- An n-state machine M can be decomposed into two independent components operating in parallel if and only if there exist two nontrivial closed partitions π_1 and π_2 on the states of M such that $\pi_1 \cdot \pi_2 = \pi(0)$. This decomposition requires a minimal number (i.e., $\lceil \log_2 n \rceil$) of state variables if and only if

$$\lceil \log_2 \#(\pi_1) \rceil + \lceil \log_2 \#(\pi_2) \rceil = \lceil \log_2 n \rceil.$$

Example Consider the machine M_2 given in Table 12.3. It can be shown that M_2 has seven closed partitions, which are listed in Fig. 12.4. Since M_2 has eight states, three state variables are needed for an assignment. The existence of the closed partition π_5 suggests that M_2 can be realized as two component machines connected in series. The predecessor component has two state variables, y_1 and y_2, which are assigned to the blocks of π_5 and,

Table 12.3 Machine M_2

	NS		
PS	$x=0$	$x=1$	z
A	H	B	0
B	F	A	0
C	G	D	0
D	E	C	1
E	A	C	0
F	C	D	0
G	B	A	0
H	D	B	0

$$\pi_0 = \{\overline{A;B;C;D;E;F;G;H}\} = \pi(0),$$
$$\pi_1 = \{\overline{A,B,C,D;E,F,G,H}\},$$
$$\pi_2 = \{\overline{A,D,E,H;B,C,F,G}\},$$
$$\pi_3 = \{\overline{A,D;B,C,F,G;E,H}\},$$
$$\pi_4 = \{\overline{A,D,E,H;B,C;F,G}\},$$
$$\pi_5 = \{\overline{A,D;B,C;E,H;F,G}\},$$
$$\pi_6 = \{\overline{A,B,C,D,E,F,G,H}\} = \pi(I).$$

Fig. 12.4 Closed partitions for M_2.

consequently, are independent of y_3, while the successor component has a single variable, y_3, which distinguishes the states in the blocks of π_5.

Maximal reduction in the dependency of state variables would be achieved if we could find three two-block closed partitions whose product is zero. In such a case, each state variable would be independent of the remaining two variables and the machine would be realized as a parallel connection of three component machines. It is evident, however, from the list of nontrivial closed partitions of M_2 that only two two-block partitions can be found, namely, π_1 and π_2. In fact, since each nontrivial closed partition is greater than π_5, no combination of closed partitions can be found whose product is zero. Therefore, we must select a partition τ such that

$$\pi_1 \cdot \pi_2 \cdot \tau = \pi(0).$$

One possible such partition is

$$\tau = \{\overline{A,D,G,H};\overline{C,D,E,F}\}.$$

Assigning y_1 to distinguish the blocks of π_1, y_2 to distinguish the blocks of π_2, and y_3 to distinguish the blocks of τ results in the assignment given in Table 12.4. Clearly, y_1 and y_2, which are assigned to the blocks of closed partitions, will be self-dependent, while y_3, which is assigned to the blocks

Table 12.4 Excitation and output table for M_2

		$Y_1Y_2Y_3$		
	$y_1y_2y_3$	$x=0$	$x=1$	z
A	000	100	010	0
B	010	111	000	0
C	011	110	001	0
D	001	101	011	1
E	101	000	011	0
F	111	011	001	0
G	110	010	000	0
H	100	001	010	0

of τ, will be a function of the external input and all three state variables. The logic equations derived from Table 12.4 are

$$
\begin{aligned}
Y_1 &= x'y_1', \\
Y_2 &= x'y_2 + xy_2', \\
Y_3 &= xy_3 + x'y_1'y_2y_3' + y_1'y_2'y_3 + y_1y_2y_3 + x'y_1y_2'y_3', \\
z &= y_1'y_2'y_3.
\end{aligned}
$$

The corresponding schematic diagram is shown in Fig. 12.5.

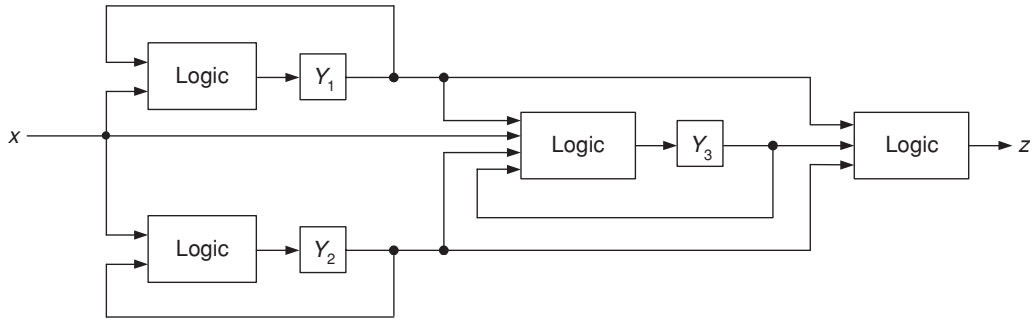


Fig. 12.5 Schematic diagram for M_2.

12.3 The lattice of closed partitions

Closed partitions have been shown to play a significant role in the state-assignment problem and in determining the dependency of the state variables. Therefore we will present a method for generating these partitions and will investigate their properties.

Theorem 12.2 *The product $\pi_1 \cdot \pi_2$ and the sum $\pi_1 + \pi_2$ of two closed partitions on the set of states of M are also closed.*

Proof Let π_1 and π_2 be two closed partitions on the states of M. We will show that the partition $\pi_1 \cdot \pi_2$ is also closed, leaving the proof that $\pi_1 + \pi_2$ is closed as an exercise to the reader.

Let B be an arbitrary block of $\pi_1 \cdot \pi_2$. Since B is the intersection of some blocks B_1 of π_1 and B_2 of π_2, B is contained in both B_1 and B_2. Since π_1 and π_2 are closed, the I_k-successor of B is also contained within some block $I_k B_1$ of π_1 and some block $I_k B_2$ of π_2, where $I_k B_i$ is the I_k-successor of B_i. Therefore $I_k B$ is contained within the intersection $I_k B_1 \cdot I_k B_2$. However, the intersection $I_k B_1 \cdot I_k B_2$ is contained in a block of $\pi_1 \cdot \pi_2$ and, consequently, $I_k B$ is contained in a block of $\pi_1 \cdot \pi_2$. Therefore, $\pi_1 \cdot \pi_2$ is closed. ◇

From this theorem, it follows that to each pair of closed partitions π_1 and π_2 there corresponds a *least upper bound (lub)* $\pi_1 + \pi_2$ and a *greatest lower bound (glb)* $\pi_1 \cdot \pi_2$. Consequently, the set of closed partitions on the states of a machine is closed under the $+$ and $\cdot$ binary operations and, therefore, forms a lattice (by Definition 2.2 in Section 2.4). This lattice is referred to as the *π-lattice*.

Let $\pi_{S_i S_j}$ be the *smallest* closed partition containing S_i and S_j in one block. We shall subsequently refer to the placing of S_i and S_j in one block as *identifying* them. To determine $\pi_{S_i S_j}$, we first identify S_i and S_j. This identification implies that we must also identify the successors $I_k S_i$ and $I_k S_j$ for every input symbol I_k in I. States $I_k S_i$ and $I_k S_j$ are said to be *implied* by S_i and S_j. Whenever a state S_i is identified with S_j and S_k, the transitive law must be applied in such a way that (S_i, S_j, S_k) are placed in the same block of π. If we repeat the above procedure and find the smallest closed partition $\pi_{S_i S_j}$ for every pair of states $S_i S_j$, we obtain a set of partitions known as the *basic* partitions. The π-lattice can now be obtained in two steps:

1. for every pair of states $S_i S_j$, obtain $\pi_{S_i S_j}$;
2. obtain all possible sums of basic partitions.

Since every closed partition can be shown (see Problem 12.5) to be the sum of one or more basic partitions, the above procedure indeed generates the set of all closed partitions.

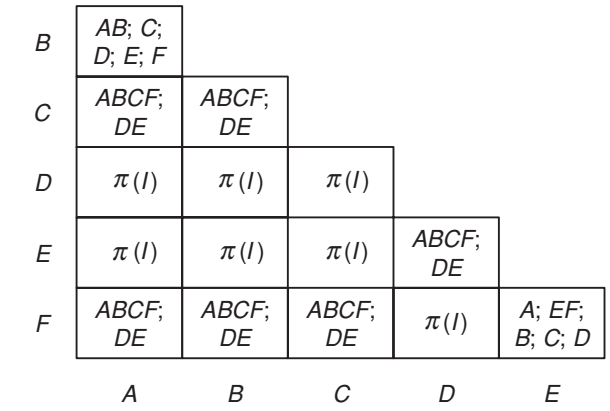
As an illustration, we shall determine the π-lattice of the machine M_3 shown in Table 12.5. The table in Fig. 12.6*a* shows the possible initial identifications and their implications. Within the cell in row S_i, column S_j, we write the identifications implied by the initial identification of S_i and S_j. For example, if we start by identifying the states A, B, we find that no other pair of states is implied. Consequently, the partition $\{\overline{A, B}; \overline{C}; \overline{D}; \overline{E}; \overline{F}\}$ is closed. We continue by identifying A, C, which, in turn, implies A, B and D, E. These implications may be described as

$$A, C \rightarrow A, B; D, E.$$

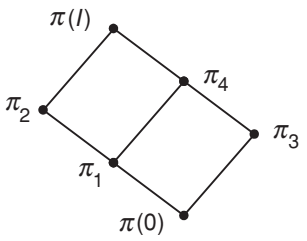
Table 12.5 Machine M_3

	NS	
PS	$x = 0$	$x = 1$
A	E	B
B	E	A
C	D	A
D	C	F
E	F	C
F	E	C

Fig. 12.6 Construction of the π-lattice of M_3.



(*a*) Derivation of basic partitions.



$\pi_0 = \pi(0)$
$\pi_1 = \{\overline{A,B}; \overline{C}; \overline{D}; \overline{E}; \overline{F}\}$
$\pi_2 = \{\overline{A,B,C,F}; \overline{D,E}\}$
$\pi_3 = \{\overline{A}; \overline{B}; \overline{C}; \overline{D}; \overline{E,F}\}$
$\pi_4 = \{\overline{A,B}; \overline{C}; \overline{D}; \overline{E,F}\}$
$\pi_5 = \pi(I)$

(*b*) π-lattice.

It is already known that the identification of A, B does not imply any other pair. Hence, we need to check only the implications due to D, E. From the state table we find that D, E implies C, F. Since A, C and C, F are identified, the transitive law must be applied to yield A, C, F. This process is thus summarized as follows:

$$A, C \rightarrow A, B; D, E \rightarrow A, C, F; A, B; D, E \rightarrow A, B, C, F; D, E.$$

The entire table is completed in a similar manner. Many shortcuts are possible. For example, while identifying B, D, the pair A, F is implied. However, since the implications which result from the identification of A, F have already been determined, it becomes immediately evident that the identification of B, D implies the identity partition, i.e.,

$$B, D \rightarrow C, E; A, F \rightarrow A, B, C, F; D, E; C, E \rightarrow \pi(I).$$

The next step in the procedure is to determine the remaining (nonbasic) closed partitions. This is done by computing the sums of pairs of basic partitions to obtain "second-level" partitions and then using only pairs of "second-level" partitions to obtain "third-level" partitions, and so on. For the machine M_3, the basic partitions are

$$\pi_1 = \{\overline{A, B}; \overline{C}; \overline{D}; \overline{E}; \overline{F}\},$$
$$\pi_2 = \{\overline{A, B, C, F}; \overline{D, E}\},$$
$$\pi_3 = \{\overline{A}; \overline{B}; \overline{C}; \overline{D}; \overline{E, F}\}.$$

The only sum that yields a nontrivial closed partition is

$$\pi_4 = \pi_1 + \pi_3 = \{\overline{A, B}; \overline{C}; \overline{D}; \overline{E, F}\}.$$

The π-lattice for the machine M_3 is shown in Fig. 12.6*b*.

12.4 Reduction of the output dependency

So far, attention has been focused on reducing the dependency of state variables. In assigning the states of these variables to the blocks of a closed partition, we have a considerable amount of freedom. It is our aim in the following discussion to show how this freedom can be used to obtain simpler output circuits with reduced dependencies. The problem is illustrated by considering two possible assignments for the machine M_4 shown in Table 12.6.

Machine M_4 possesses the closed partition $\pi = \{\overline{A, B}; \overline{C, D}\}$. To obtain a state assignment, we are looking for a partition τ such that $\pi \cdot \tau = \pi(0)$. The assignments α and β shown in Table 12.7 correspond, respectively, to the partitions $\tau_a = \{\overline{A, C}; \overline{B, D}\}$ and $\tau_b = \{\overline{A, D}; \overline{B, C}\}$. The state variables and

Table 12.6 Machine M_4

	NS		z	
PS	$x = 0$	$x = 1$	$x = 0$	$x = 1$
A	B	D	1	0
B	A	C	0	1
C	D	A	0	1
D	C	B	1	0

Table 12.7 Two possible assignments for machine M_4

	y_1y_2
A	00
B	01
C	10
D	11

(a) Assignment α

	y_1y_2
A	00
B	01
C	11
D	10

(b) Assignment β

output function corresponding to assignment α are as follows:

$$\begin{aligned} Y_1 &= x'y_1 + xy_1', \\ Y_2 &= x'y_2' + y_1'y_2' + xy_1y_2, \\ z &= x'y_1'y_2' + x'y_1y_2 + xy_1'y_2 + xy_1y_2'. \end{aligned}$$

The number of transistors required for a two-level NAND–NAND CMOS realization of these functions is 64.

For assignment β, we obtain

$$\begin{aligned} Y_1 &= x'y_1 + xy_1', \\ Y_2 &= x'y_2' + xy_1'y_2 + y_1y_2', \\ z &= x'y_2' + xy_2. \end{aligned}$$

The realization of these functions requires only 40 transistors.

Evidently, the reduction in circuit complexity is the outcome of the decrease in the dependency of the output function. While in assignment α the output depends on x, y_1, and y_2, in assignment β it is independent of y_1. Although such a reduction in the dependency of the output does not always ensure simpler output circuits, in most cases it does tend to decrease the complexity of the circuit. Our aim, therefore, is directed towards obtaining assignments which reduce the dependencies of the output logic.

Definition 12.2 A partition λ_o on the states of a machine M is said to be *output-consistent* if, for every block of λ_o and every input symbol, all the states contained in the block have the same output symbols.

Example The partition $\lambda_o = \tau_b = \{\overline{A, D}; \overline{B, C}\}$ is an output-consistent partition of the machine M_4.

Let M have n states to which we assign k variables, where $k = \lceil \log n \rceil$. Let $r = \lceil \log_2 \#(\lambda_o) \rceil$ variables be assigned to the blocks of M's output-consistent partition λ_o. Because λ_o is output-consistent, the output symbols associated with the blocks of λ_o can be computed from these r variables, independently

of the remaining $k - r$ variables assigned to the states in the blocks of λ_o. Consequently, we arrive at the following general result.

- The existence of an output-consistent partition λ_o on the states of a sequential machine M implies that there exists an assignment for M such that the outputs depend, at most, on the external inputs and on the variables assigned to the blocks of λ_o.

This result can be generalized as follows. Let $\Theta = \{\tau_1, \tau_2, \ldots, \tau_k\}$ be the set consisting of partitions induced by the state variables $y_1, y_2, \ldots, y_k$. Let $\lambda_{o1}, \lambda_{o2}, \ldots, \lambda_{om}$ be the output-consistent partitions induced by the outputs $z_1, z_2, \ldots, z_m$. If, for some subset Q of Θ,

$$\lambda_{oi} \geq \prod_{j \in Q} \tau_j$$

then z_i is a function of the external input x and the variables assigned to the partitions contained in Q.

Example In the machine M_4, $\lambda_o = \lambda_{o1} = \{\overline{A, D}; \overline{B, C}\}$. Since y_2 has been assigned to λ_o in assignment β, the output z depends only on this variable and is independent of y_1.

In assignment β we obtained a reduction in the dependency of y_1 and (simultaneously) of the output z. This is possible since $\pi \cdot \lambda_o = \pi(0)$. In general, however, we cannot efficiently obtain a complete assignment on the basis of any arbitrary closed partition π and any output-consistent partition λ_o. For example, if $\pi \cdot \lambda_o = \pi(0)$ but $\lceil \log_2 \#(\pi) \rceil + \lceil \log_2 \#(\lambda_o) \rceil > \lceil \log_2 n \rceil$ then an assignment can be obtained in which the outputs depend on $\lceil \log_2 \#(\lambda_o) \rceil$ variables and these $\lceil \log_2 \#(\pi) \rceil$ variables are independent of the remaining ones. However, such an assignment is not minimal since it requires extra variables. For example, if $\pi = \{\overline{A, B}; \overline{C, D}; \overline{E, F}; \overline{G, H}\}$ while $\lambda_o = \{\overline{A, C}; \overline{B, E}; \overline{D, G}; \overline{F, H}\}$ then $\pi \cdot \lambda_o = \pi(0)$ but $\log_2 4 + \log_2 4 = 4$. If we use only π or only λ_o, we can obtain an assignment with only three variables. It should be noted that, while λ_o simplifies the output circuit, the additional variables (the fourth one in the above case), which are not assigned to any closed partition, may add a significant amount of logic to the overall circuit. Consequently, we have two different requirements: to make an assignment based on an output-consistent partition λ_o and, at the same time, to reduce the dependencies of the state variables, i.e., to assign the variables to the blocks of a closed partition π. These two requirements often conflict. Various approaches have been tried in attempts to solve this problem (see, for example, [10]). This may require some trial and error.

12.5 Input independency and autonomous clocks

Some machines can be constructed from two components: one input-independent and the other input-dependent. Our aim in this section is to determine necessary and sufficient conditions for the existence of state assignments that result in such a structure.

Definition 12.3 A partition λ_i on the states of a machine M is said to be *input-consistent* if, for every state S_i of M and all input symbols $I_1, I_2, \ldots, I_p$, the next states $I_1S_i, I_2S_i, \ldots, I_pS_i$ are in the same block of λ_i.

Example Consider the machine M_5 shown in Table 12.8. State A implies the identification of states C and D. Similarly, the identification of E and F is implied by state C, while the identification of A and B is implied by state E. Thus, the smallest input-consistent partition for M_5 is $\lambda_i = \{\overline{A, B}; \overline{C, D}; \overline{E, F}\}$. Clearly, any partition that contains λ_i is also input-consistent. Unless otherwise indicated, λ_i will subsequently designate the smallest input-consistent partition.

Table 12.8 Machine M_5

	NS		z	
PS	$x=0$	$x=1$	$x=0$	$x=1$
A	D	C	0	1
B	C	D	0	0
C	E	F	0	1
D	F	F	0	0
E	B	A	0	1
F	A	B	0	0

Since the successor relationships between the blocks of λ_i are independent of the inputs, the $\lceil \log_2 \#(\lambda_i) \rceil$ variables assigned to distinguish the blocks of λ_i are input-independent. If, in addition to λ_i, a machine M possesses a closed partition π such that $\pi \geq \lambda_i$ then, for a given state S_j and every input symbol $I_1, I_2, \ldots, I_p$ in I, the next states, $I_1S_j, I_2S_j, \ldots, I_pS_j$, must be in the same block of λ_i and, therefore, in the same block of π as well. Consequently, for a given initial state, the block of π in which the state of M is contained after any finite input sequence depends only on the initial block and on the length of the sequence. This property may be summarized as follows.

- The existence of a closed partition π and a nontrivial input-consistent partition λ_i on the states of M, where $\pi \geq \lambda_i$, is a necessary and sufficient condition for the existence of an assignment for M such that the $\lceil \log_2 \#(\pi) \rceil$ variables assigned to the blocks of π are independent of the input and of the remaining state variables.

A component machine whose output at any time is independent of the input is called an *autonomous clock*. If M possesses an input-consistent partition λ_i and several closed partitions, each greater than or equal to λ_i, then the autonomous clock corresponding to the smallest such closed partition is referred to as the *maximal autonomous clock*.

Example For M_5, the input-consistent partition $\lambda_i = \{\overline{A, B}; \overline{C, D}; \overline{E, F}\}$ is closed. The output-consistent partition is $\lambda_o = \{\overline{A, C, E}; \overline{B, D, F}\}$. Since $\pi = \lambda_i$ and $\pi \cdot \lambda_o = \pi(0)$ the assignment and logic equations in Table 12.9 result. The schematic diagram corresponding to this assignment is shown in Fig. 12.7. It clearly displays the existence of an autonomous clock as well as the reduction in the dependency of z due to λ_o. The external clock has not been shown but is implicit. In fact, it triggers the autonomous clock and causes it to change states.

Table 12.9 Assignments and equations for M_5

	$y_1y_2y_3$
A	000
B	001
C	010
D	011
E	100
F	101

(*a*) Assignment

$$Y_1 = y_2$$
$$Y_2 = y_1'y_2'$$
$$Y_3 = xy_2 + xy_3 + x'y_2'y_3' + y_2y_3$$
$$z = xy_3'$$

(*b*) Logical equations

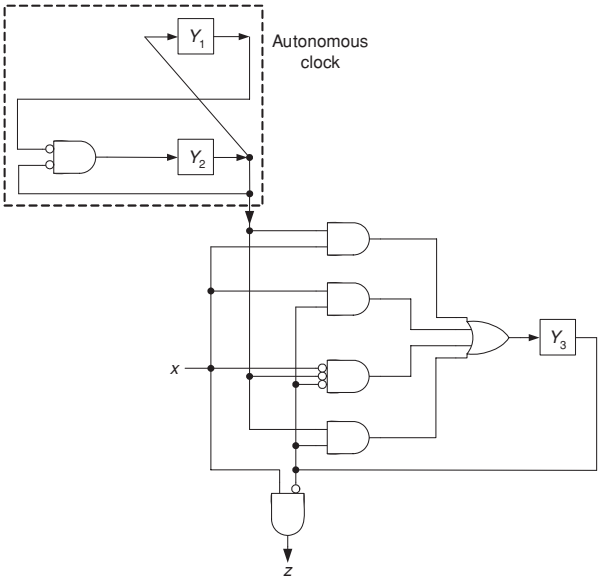


Fig. 12.7 Realization of M_5.

It is easy to show that if M is a strongly connected machine then any component induced by a closed partition on the states of M is also strongly connected. Hence, the autonomous clock of a strongly connected machine is also strongly connected and, furthermore, it is a periodic machine. To find the period p of the autonomous clock, suppose that the machine M possesses a closed partition π such that $\pi \geq \lambda_i$. The clock has $\#(\pi)$ states and, therefore, during $\#(\pi) + 1$ time units, it must pass at least twice through one of the states. Thus, the period p is less than or equal to $\#(\pi)$.

Example The maximal autonomous clock of machine M_5 is determined from the partition $\pi = \lambda_i$, where

$$\pi = \{\overline{A, B}; \overline{C, D}; \overline{E, F}\} = \{\overline{\alpha}; \overline{\beta}; \overline{\gamma}\}.$$

In the state table of M_5, let us denote the blocks (A, B), (C, D), and (E, F) by α, β, and γ, respectively. The graph describing the block-successor relationships of π yields the state diagram of the maximal autonomous clock, as shown in Fig. 12.8. From the graph it is clear that the period p of the clock is 3.

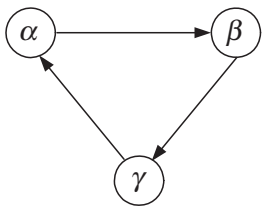


Fig. 12.8 The autonomous clock of machine M_5.

12.6 Covers, and the generation of closed partitions by state splitting

The correlation between closed partitions and the existence of assignments with self-dependent and autonomous subsets have been established in the preceding sections. These assignments have been shown to yield simpler circuits and affect a circuit's structure. Many machines, however, do not possess such partitions and therefore cannot be implemented with independent components. Our objective in this section is to develop a method that will enable us to generalize the preceding structure theory and, by allowing the classification of the states into nondisjoint subsets, to augment a machine that does not possess any closed partition into an equivalent machine that does possess such partitions. Such an augmentation is achieved by splitting some states of the original machine. The basic tool in this procedure is the implication graph, which will be defined shortly.

Table 12.10 Machine M_6

	NS		z	
PS	$x=0$	$x=1$	$x=0$	$x=1$
A	A	B	0	1
B	C	B	0	0
C	A	C	0	0

Table 12.11 Machine M_6'

	NS		z	
PS	$x=0$	$x=1$	$x=0$	$x=1$
A	A	B	0	1
B	C'	B	0	0
C'	A	C''	0	0
C''	A	C''	0	0

Covers

To illustrate the basic ideas, consider the machine M_6 shown in Table 12.10. It can be verified that no closed partition exists for this machine and therefore it would appear that it cannot be decomposed in any manner. Consider next the machine M_6' (Table 12.11), which is reducible to machine M_6 since the states C' and C'' are equivalent. Machine M_6' possesses the closed partition $\pi = \{\overline{A, C'}; \overline{B, C''}\}$. If we choose a partition $\tau = \{\overline{A, B}; \overline{C', C''}\}$ such that $\pi \cdot \tau = \pi(0)$, and if we assign y_1 and y_2 to the blocks of π and τ, respectively, then the following equations result:

$$\begin{aligned} Y_1 &= x, \\ Y_2 &= xy_2 + x'y_1y_2', \\ z &= xy_1'y_2'. \end{aligned}$$

Clearly, machine M_6' is realizable as a serial connection of a predecessor component (Y_1) and a successor component (Y_2). Such a decomposition of machine M_6' is also a valid realization of the equivalent machine M_6, although the latter machine does not possess any closed partition. If we work backward from machine M_6' to M_6, we observe that the closed partition $\pi = \{\overline{A, C'}; \overline{B, C''}\}$ becomes equal to $\{\overline{A, C}; \overline{B, C}\}$ when the two equivalent states C' and C'' are merged. Although this collection of subsets covers all the states and is closed with respect to the states of M_6, it does not constitute a partition since its blocks are not disjoint. In order to cover such situations it becomes necessary to generalize the structure theory and to define sets consisting of overlapping subsets of states.

Table 12.12 State transitions of the predecessor component in the serial decomposition of M_6

	$x = 0$	$x = 1$
P	P	Q
Q	P	Q

A collection φ of subsets, whose set union is S, such that no subset is included in another subset in the collection, is referred to as a *cover* on set S. The subsets are called the *blocks* of φ. The cover φ on the set of states of a machine M is said to be *closed* if, for every two states S_i and S_j which are in the same block of φ and any input symbol I_k in I, the states $I_k S_i$ and $I_k S_j$ are in a common block of φ. The number of blocks in φ and the number of elements in the largest block of φ are denoted $\#(\varphi)$ and $\rho(\varphi)$, respectively.

Example The covers $\{\overline{A, C}; \overline{B, C}\}$ and $\{\overline{A, B}; \overline{A, C}; \overline{B, C}\}$ on the set of states of M_6 are closed.

If we denote subsets (AC) and (BC) by P and Q, respectively, we obtain the successor relationships given in Table 12.12. Since the predecessor machine in the serial connection of M_6 distinguishes the blocks of φ, the successor relationships of Table 12.12 define uniquely the state transitions of the predecessor component.

In order to be able to decompose machines that do not possess any closed partition, it is necessary either to generalize the results of the previous sections to include covers or develop a method whereby any such machine can be augmented to an equivalent machine that has one or more closed partitions and is, therefore, decomposable. The approach taken in this section is the latter.

The implication graph

The main difference between the machines M_6 and M_6' is that state C of M_6 has been split into states C' and C'' in M_6'. In general, state S_i is said to be *split* into states S_i' and S_i'' if (i) the output symbols of S_i' and S_i'' are exactly the same as those of S_i and (ii) for every I_k in I, states $I_k S_i'$ and $I_k S_i''$ are identical to $I_k S_i$, except where "primes" are necessary, as will be shown later.

An *implication graph* is a directed graph, with vertices representing subsets of the set of states of a machine M. Each subset consists of states to be identified in the state table of M or which are implied by previously identified subsets of states. The arc labeled I_k represents the transition from one subset of states $(S_i, S_j, \ldots)$ to the subset consisting of the I_k-successors $(I_k S_i, I_k S_j, \ldots)$.

Definition 12.4 A *closed* implication graph is a subgraph of an implication graph such that: (i) for every vertex in the subgraph all outgoing arcs and their terminating vertices also belong to the subgraph; and (ii) every state of M is represented by at least one vertex.

From the definition of the implication graph for a given machine M, it is evident that the collection of subsets associated with the vertices of the closed graph constitutes a closed cover on the set of states of M. From now on, we shall consider implication graphs whose vertices represent only pairs of states. It will be shown later that such graphs provide the necessary information regarding all closed covers.

An implication graph is constructed in the following manner. Identify any pair of states S_i and S_j and assign (S_i, S_j) to some initial vertex. For each input symbol I_k, draw an arc from the vertex (S_i, S_j) to the vertex that represents the successors $(I_k S_i, I_k S_j)$. Repeat this process for all the vertices implied by the initial identification until no new vertex is generated.

If M is strongly connected, an initial identification of any pair of states will result in a closed graph. If, however, M is not strongly connected then the closed graph might have to be constructed from two or more disjoint subgraphs, that is, another pair of states not implied by $(S_i S_j)$ must be identified, its successors determined, and so on.

Example To construct the implication graph for the machine M_6, start by identifying the pair of states (A, B). This identification implies the identification of (A, C), which in turn implies (B, C). The graph, which is closed, is shown in Fig. 12.9. It is evident that the subgraph enclosed by the broken lines is also closed, since it satisfies Definition 12.4.

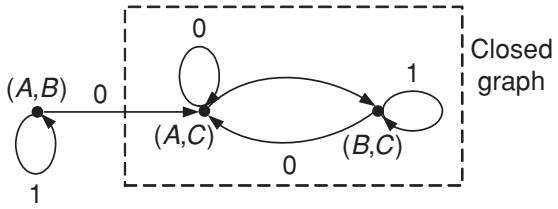


Fig. 12.9 Implication graph for M_6.

The general procedure for augmenting an arbitrary machine M into an equivalent machine M' that possesses one or more closed partitions can now be summarized as follows.

1. Construct the implication graph of the given machine M.
2. From the implication graph, choose a closed subgraph with a minimal number of vertices. This subgraph yields a closed cover φ on M. If any state S_i

is represented by more than one vertex, relabel S_i in the first vertex as S_i', in the second vertex as S_i'', and so on.

3. For each S_i that has been replaced by $S_i', S_i'', \ldots$, split the corresponding state in M's state table.
4. Modify the entries of the new state table by inserting the necessary primes. An entry S_p in row S_i, column I_k, is changed to S_p' if S_i is represented by some vertex (S_i, S_j) and the I_k-successor vertex is (S_p', S_q).

Example In the implication graph of Fig. 12.9, state C appears in two vertices and thus is split into C' and C'', as shown in Table 12.11. The partition $\pi = \{\overline{A, C'}; \overline{B, C''}\}$, whose blocks correspond to subsets represented by vertices of the implication graph, is clearly closed.

In general, a partition π whose blocks correspond to subsets represented by vertices of the closed implication graph is closed with respect to the set of states of the augmented machine M'. This partition has a finite number of blocks, since $(n-1)n/2$ is the total number of distinct pairs of states. The closed implication graph actually describes the successor relationship of the blocks of π graphically and, consequently, represents the state diagram of the predecessor component in a possible serial realization of M'. The *implication table*, which is the tabular representation of the implication graph, is therefore the state table of the predecessor component. The implication table that corresponds to the closed graph of Fig. 12.9 was derived earlier and is shown in Table 12.12.

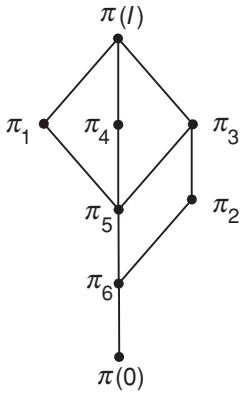
From the foregoing procedure it follows that *corresponding to every finite-state machine M, there exists at least one equivalent finite-state machine M' that possesses a closed partition and is therefore serially decomposable*. It should be emphasized, however, that such decompositions are not necessarily the most economical way of realizing a machine. In fact, for an n-state machine, the closed cover may have up to $(n-1)n/2$ blocks, which means that the predecessor component will have more states than the original machine. The primary case of practical interest is that in which none of the components in the decomposition is equal to or greater than the original machine. This condition is satisfied whenever the number of vertices in the closed implication graph is smaller than n.

In the foregoing discussion, attention has been focused primarily on uniform closed covers containing two states per block. The remaining covers can be determined from this set of basic covers by obtaining all possible sums in a manner analogous to the method of generating the set of closed partitions. The preceding techniques can be extended easily to blocks of any size and of uniform, as well as nonuniform, covers.

Table 12.13 Machine M_7

	NS		z	
PS	$x=0$	$x=1$	$x=0$	$x=1$
A	B	C	0	0
B	A	F	1	1
C	F	E	1	0
D	F	E	1	1
E	G	D	0	0
F	D	B	0	0
G	E	F	1	0

Fig. 12.10 The π-lattice for M_7.



$$\pi(I) = \{\overline{A,B,C,D,E,F,G}\}$$
$$\pi_1 = \{\overline{A,C,D,E};\ \overline{B,F,G}\}$$
$$\pi_2 = \{\overline{A,G};\ \overline{B,E};\ \overline{C,D,F}\}$$
$$\pi_3 = \{\overline{A,B,E,G};\ \overline{C,D,F}\}$$
$$\pi_4 = \{\overline{A,E,F};\ \overline{B,C,D,G}\}$$
$$\pi_5 = \{\overline{A,E};\ \overline{B,G};\ \overline{C,D};\ \overline{F}\}$$
$$\pi_6 = \{\overline{A};\ \overline{B};\ \overline{C,D};\ \overline{E};\ \overline{F};\ \overline{G}\}$$
$$\pi(0) = \{\overline{A};\ \overline{B};\ \overline{C};\ \overline{D};\ \overline{E};\ \overline{F};\ \overline{G}\}$$

An application of state splitting to parallel decomposition

A machine M_7 and its π-lattice are given in Table 12.13 and Fig. 12.10, respectively. In addition to these closed partitions, M_7 possesses an output-consistent partition λ_o and an input-consistent partition λ_i, namely,

$$\lambda_o = \{\overline{A, E, F}; \overline{B, D}; \overline{C, G}\},$$
$$\lambda_i = \{\overline{A, E, F}; \overline{B, C, D, G}\} = \pi_4.$$

Our aim is to obtain a parallel decomposition of M_7. A brief inspection of the π-lattice reveals that no such decomposition is possible, since no two nontrivial closed partitions exist such that $\pi_i \cdot \pi_j = \pi(0)$ (the subset (C, D) is common to all nontrivial partitions). Consequently, it becomes necessary to check whether there exist any closed covers that yield a parallel decomposition.

The implication graph, when started by the identification of (A, B), is given in Fig. 12.11. From the closed graph, we obtain the closed cover

$$\varphi = \{\overline{A, G}; \overline{B, E}; \overline{C, F}; \overline{D, F}\}.$$

The corresponding augmented machine M_7' is given in Table 12.14.

Table 12.14 Machine M_7'

	NS		z	
PS	$x=0$	$x=1$	$x=0$	$x=1$
A	B	C	0	0
B	A	F''	1	1
C	F''	E	1	0
D	F''	E	1	1
E	G	D	0	0
F'	D	B	0	0
F''	D	B	0	0
G	E	F'	1	0

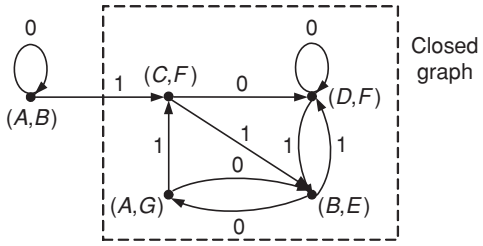


Fig. 12.11 The implication graph for M_7.

In general, for every closed partition π on M, a corresponding closed partition π' on M' can be obtained by placing the states S_i', S_i'', etc., in π' for every split state S_i in π. The closed partitions of machine M_7', which may be used to achieve a parallel decomposition, are

$$\pi = \{\overline{A, G}; \overline{B, E}; \overline{C, F'}; \overline{D, F''}\},$$
$$\pi_4' = \{\overline{A, E, F', F''}; \overline{B, C, D, G}\},$$
$$\pi_3' = \{\overline{A, B, E, G}; \overline{C, D, F', F''}\}.$$

In addition, the augmented machine possesses the following output-consistent and input-consistent partitions:

$$\lambda_o' = \{\overline{A, E, F', F''}; \overline{B, D}; \overline{C, G}\},$$
$$\lambda_i' = \pi_4'.$$

From this set of partitions, the following observations can be made:

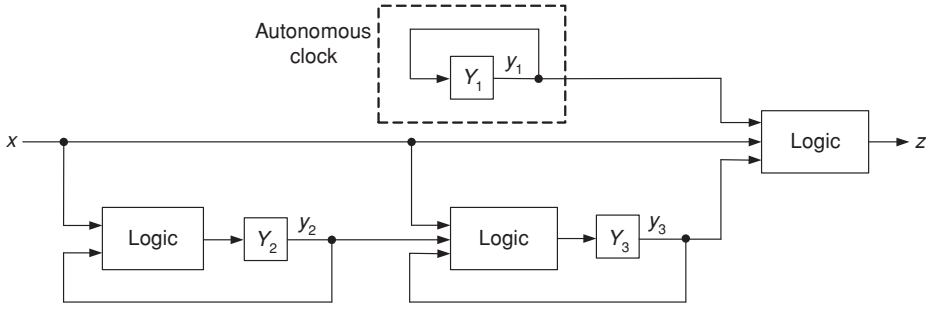
1. The product $\pi \cdot \pi_4' = \pi(0)$, which implies that a parallel decomposition is possible.
2. The component machine corresponding to π_4' consists of a single variable, y_1. It is an autonomous clock since $\pi_4' = \lambda_i'$.
3. Because each block of π_3' contains exactly two blocks of π, we may assign y_2 to the blocks of π_3' and thus make it independent of the value of y_3.

Fig. 12.12 Decomposition of M_7'.

	$y_1y_2y_3$
A	000
B	101
C	110
D	111
E	001
F'	010
F''	011
G	100

$$Y_1 = y_1'$$
$$Y_2 = x'y_2 + xy_2'$$
$$Y_3 = y_2 + xy_3 + x'y_3'$$
$$z = x'y_1 + y_1y_3$$

(*a*) Assignment and logic equations.



(*b*) Schematic diagram.

4. The variable y_3 must be assigned to the blocks of a partition τ such that $\pi_3' \cdot \tau = \pi$. The partition $\tau = \{\overline{A, C, F', G}; \overline{B, D, E, F''}\}$ satisfies this condition.
5. The product $\tau \cdot \pi_4' = \{\overline{A, F'}; \overline{B, D}; \overline{C, G}; \overline{E, F''}\}$ is smaller than λ_o'; consequently, the output z will be a function of only y_1 and y_3.

The assignment and logic equations resulting from the preceding observations are shown in Fig. 12.12*a*. The schematic diagram is shown in Fig. 12.12*b*.

12.7 Information flow in sequential machines

In the previous sections we have dealt mainly with serial and parallel decompositions. Of course, there are more complex structures, and our aim in this section is to define them and determine the conditions under which they exist. The main tool for accomplishing this task is the partition pair. It will be shown that the problem of finding state assignments leading to specified machine structures is equivalent to the problem of finding an appropriate set of partition pairs and determining their properties.

Table 12.15 Machine M_8

	NS				
PS	x_1x_2 00	01	11	10	z
A	A	C	D	F	0
B	C	B	F	E	0
C	A	B	F	D	0
D	E	F	B	C	0
E	E	D	C	B	0
F	D	F	B	A	1

Table 12.16 Two possible assignments for M_8

	$y_1y_2y_3$
A	000
B	010
C	011
D	111
E	100
F	110

(a) Assignment α

	$y_1y_2y_3$
A	000
B	011
C	010
D	110
E	100
F	111

(b) Assignment β

Introduction

The machine M_8 shown in Table 12.15 possesses two closed partitions: $\pi_1 = \{\overline{A, B, C}; \overline{D, E, F}\}$ and $\pi_2 = \{\overline{A, E}; \overline{B, F}; \overline{C, D}\}$, where $\pi_1 \cdot \pi_2 = \pi(0)$. Consequently, M_8 can be decomposed into two parallel components, as shown by assignment α in Table 12.16a. The corresponding logic equations for the state variables are

$$Y_1 = x_1'y_1 + x_1y_1' = f_1(x_1, y_1),$$
$$Y_2 = x_2 + x_1y_2' + x_1y_3 + x_1'y_2y_3' = f_2(x_1, x_2, y_2, y_3),$$
$$Y_3 = x_1'x_2'y_2y_3' + x_2y_2' + x_1x_2'y_2y_3 = f_3(x_1, x_2, y_2, y_3).$$

The two-level NAND–NAND CMOS realization of the above equations requires 60 transistors, and the functional dependencies are such that two of the next-state variables (Y_2 and Y_3) each depend on two of the present-state variables (y_2 and y_3).

Next, we examine assignment β in Table 12.16b, which yields the following equations:

$$Y_1 = x_1'y_1 + x_1y_1' = f_1(x_1, y_1),$$
$$Y_2 = x_2 + x_1'y_3 + x_1y_3' = f_2(x_1, x_2, y_3),$$
$$Y_3 = x_2y_2 + x_1x_2'y_2' = f_3(x_1, x_2, y_2).$$

The two-level realization of these equations requires only 40 transistors. This reduction in the number of transistors has been accomplished by reducing the functional dependencies of the variables, since each next-state variable now depends on just a single present-state variable. Evidently, this type of reduced dependency (which actually contains "cross dependencies") cannot be predicted just from the closed partitions. Consequently, a more general tool is needed.

Partition pairs

In order to determine the cause of the cross dependencies obtained by assignment β, we first observe that y_1 induces π_1 while y_2 and y_3 induce the partitions $\tau(y_2) = \{\overline{A, E}; \overline{B, C, D, F}\}$ and $\tau(y_3) = \{\overline{A, C, D, E}; \overline{B, F}\}$, respectively, where $\pi_1 \cdot \tau(y_2) \cdot \tau(y_3) = \pi(0)$. Except for π_1, neither of these partitions is closed although the product $\tau(y_2) \cdot \tau(y_3) = \pi_2$ is closed. However, knowledge of the block of $\tau(y_2)$ and the input symbols is sufficient to determine uniquely the successor block contained in some block of $\tau(y_3)$; that is, successors of the blocks of $\tau(y_2)$ are contained in the blocks of $\tau(y_3)$. Similarly, it is evident that the blocks of $\tau(y_2)$ are successors of the blocks of $\tau(y_3)$.

Definition 12.5 A *partition pair* (τ, τ') on the states of a sequential machine M is an ordered pair of partitions such that, if S_i and S_j are in the same block of τ then, for every input symbol I_k in I, $I_k S_i$ and $I_k S_j$ are in the same block of τ'.

Thus τ' consists of all the successor blocks implied by τ. If $\tau = \tau'$ then τ is closed, since it contains its own successor blocks. Hence, the set of closed partitions may be viewed as a special case of the (more general) set of partition pairs.

Example The following are partition pairs on the states of M_8:

$$(\pi_1, \pi_1') = (\{\overline{A, B, C}; \overline{D, E, F}\}, \{\overline{A, B, C}; \overline{D, E, F}\}),$$
$$(\tau_1, \tau_1') = (\{\overline{A, C, D, E}; \overline{B, F}\}, \{\overline{A, E}; \overline{B, C, D, F}\}),$$
$$(\tau_2, \tau_2') = (\{\overline{A, E}; \overline{B, C, D, F}\}, \{\overline{A, C, D, E}; \overline{B, F}\}).$$

In assignment β of Table 12.16, y_1, y_2, and y_3 have been assigned to π_1', τ_1', and τ_2', respectively. Note that in this example, (τ_1', τ_1) and (τ_2', τ_2) are also partition pairs.

In general, since τ consists of the blocks we want to identify while τ' contains the implied successor blocks, it is evident that any partition τ_p' such that $\tau_p' \geq \tau'$ will also contain the successor blocks of τ. Similarly, the implied successors of any partition τ_q such that $\tau_q \leq \tau$ are smaller than those of τ and, therefore,

will be contained within the blocks of τ'. Thus, the pairs (τ_q, τ') and (τ, τ'_p) are also partition pairs on the states of M.

Example The pair $(\tau_3, \tau'_3) = (\{\overline{A, D}; \overline{B}; \overline{C, E}; \overline{F}\}, \{\overline{A, E}; \overline{B, D}; \overline{C, F}\})$ is a partition pair on M_8. The following are also partition pairs on M_8:

$$(\{\overline{A, D}; \overline{B}; \overline{C}; \overline{E}; \overline{F}\}, \{\overline{A, E}; \overline{B, D}; \overline{C, F}\}),$$
$$(\{\overline{A, D}; \overline{B}; \overline{C, E}; \overline{F}\}, \{\overline{A, E}; \overline{B, C, D, F}\}).$$

A partial ordering on partition pairs is defined in the following way. If (τ_1, τ'_1) and (τ_2, τ'_2) are partition pairs then $(\tau_1, \tau'_1) \geq (\tau_2, \tau'_2)$ if and only if $\tau_1 \geq \tau_2$ and $\tau'_1 \geq \tau'_2$. We shall now prove that if (τ_1, τ'_1) and (τ_2, τ'_2) are partition pairs on the states of a machine M then $(\tau_1 \cdot \tau_2, \tau'_1 \cdot \tau'_2)$ and $(\tau_1 + \tau_2, \tau'_1 + \tau'_2)$ are also partition pairs on the states of M and define, respectively, the glb and lub of the given partition pairs. The assertion that $(\tau_1 \cdot \tau_2, \tau'_1 \cdot \tau'_2)$ is the glb of (τ_1, τ'_1) and (τ_2, τ'_2) can be proved by observing that if S_i and S_j are contained in some block of $\tau_1 \cdot \tau_2$, then they are contained in the same block in τ_1 and in τ_2. Therefore, for every input symbol I_k, the successors $I_k S_i$ and $I_k S_j$ are also contained in the same block of τ'_1 and τ'_2 and, hence, of $\tau'_1 \cdot \tau'_2$. The assertion that $(\tau_1 + \tau_2, \tau'_1 + \tau'_2)$ is the lub of (τ_1, τ'_1) and (τ_2, τ'_2) can be proved in a similar manner. Consequently, the set of all partition pairs forms a lattice under the above partial ordering.

Definition 12.6 Let τ' be a partition on the set of states of M. Define a partition $M(\tau')$ such that $M(\tau') = \sum \tau_i$, where the sum is over all τ_i such that (τ_i, τ') is a partition pair. Similarly, define a partition $m(\tau) = \prod \tau'_i$, where the product is over all τ'_i such that (τ, τ'_i) is a partition pair. A partition pair (τ, τ') is said to be an *Mm pair* if and only if $\tau = M(\tau')$ and $\tau' = m(\tau)$.

Since the lub of two partition pairs is a partition pair it follows that $(M(\tau'), \tau')$ is a partition pair, where $M(\tau')$ is the lub of all τ_i such that (τ_i, τ') is a partition pair. In fact, $M(\tau')$ *is the largest partition the successors of whose blocks are contained in the blocks of* τ'. Similarly, since the glb of two partition pairs is a partition pair, it follows that $(\tau, m(\tau))$ is a partition pair, where $m(\tau)$ is the glb of all τ'_i such that (τ, τ'_i) is a partition pair. The partition $m(\tau)$ *is thus the smallest partition containing all the successors of the blocks of* τ. Hence, $m(\tau)$ describes the largest amount of information that can be obtained from τ regarding the next state of the machine M.

It can be shown (see Problem 12.15) that the M and m partitions possess the following properties. If τ is a partition on machine M then

$$m[M(\tau)] \leq \tau,$$
$$M[m(\tau)] \geq \tau,$$
$$M\{m[M(\tau)]\} = M(\tau),$$
$$m\{M[m(\tau)]\} = m(\tau).$$

Consequently, for every partition τ on the states of M, $\{M(\tau), m[M(\tau)]\}$ and $\{M[m(\tau)], m(\tau)\}$ are Mm pairs on the states of M.

If (λ, λ') is an Mm pair then λ is the largest partition from that we can determine λ' and, at the same time, λ' is the smallest partition that contains the successor blocks implied by λ. Thus, by enlarging λ' or by refining λ, we can obtain other partition pairs. Consequently, corresponding to every partition pair (τ, τ') there exists an Mm pair (λ, λ') such that $\lambda \geq \tau$ and $\lambda' \leq \tau'$. Clearly, the set of all Mm pairs (which is, in general, substantially smaller than the set of all partition pairs) completely characterizes the set of all partition pairs on the states of M, since any partition pair can be generated from the corresponding Mm pair, as shown above.

Information-flow inequalities

In this section we shall derive the main theorem relating the algebraic properties of partitions to the dependencies of state variables and the structure of sequential machines. We shall also show that the existence of assignments with reduced dependencies of state variables can be predicted from the set of Mm pairs associated with the machine.

Theorem 12.3 *Let the variables $y_1, y_2, \ldots, y_k$ be assigned to the states of machine M, and let $\tau(y_i)$ be the partition induced by the variable y_i, where $1 \leq i \leq k$. If the next-state variable Y_i can be computed from the external inputs and a subset P_i of variables, then*

$$\prod \tau(y_j) \leq M[\tau(y_i)],$$

where the product is taken over all $\tau(y_j)$ such that y_j is contained in subset P_i. Conversely, a sufficient condition for the existence of an assignment, in which a next-state variable Y_i depends only on the external inputs and the value of a corresponding subset P_i of state variables, is the existence of a partition pair $(\tau, \tau(y_i))$ on M such that, for each τ_i',

$$\prod \tau(y_j) \leq M[\tau(y_i)],$$

where the product is taken over all $\tau(y_j)$ such that y_j is in P_i.

Proof The blocks of the partition $\prod \tau(y_j)$ consist of all the states that have the same value of the variables contained in P_i. Recalling that Y_i depends only on variable y_j if y_j is in P_i then, for any two states S_p and S_q that are in the same block of $\prod \tau(y_j)$, and for all input symbols I_k in I, the successor states $I_k S_p$ and $I_k S_q$ are in the same block of $\tau(y_i)$. Consequently,

$$\left(\prod \tau(y_j), \tau(y_i)\right)$$

is a partition pair. However, since $M[\tau(y_i)]$ is the largest partition such that $(M[\tau(y_i)], \tau(y_i))$ is a partition pair,

$$M[\tau(y_i)] \geq \prod \tau(y_j).$$

Hence, if the next-state variable Y_i can be computed from a subset of the state variables then we must have at least as much information about the present state as is contained in $M[\tau(y_i)]$.

To prove the converse note that $(M[\tau(y_i)], \tau(y_i))$ is a partition pair and, since $\prod \tau(y_j) \leq M[\tau(y_i)]$,

$$\left(\prod \tau(y_j), \tau(y_i)\right)$$

is also a partition pair. Knowledge of the values of the variables y_j in P_i is sufficient to determine the present block of $\prod \tau(y_j)$ and, therefore (by the definition of partition pairs), it is also sufficient to determine the successor block in $\tau(y_i)$. This in turn determines the value of the next state of y_i, that is, Y_i. Thus, the theorem is proved. ◇

Returning to machine M_8 we note that $\pi_1' \cdot \tau_1' \cdot \tau_2' = \pi(0)$ and that $\pi_1 = \pi_1'$, $\tau_1' = \tau_2$, and $\tau_2' = \tau_1$. Therefore, a three-variable assignment exists such that Y_1 (which is assigned to π_1') is self-dependent while Y_2 and Y_3 (which are assigned to τ_1' and τ_2') can be computed from y_3 and y_2, respectively. The above arguments lead to assignment β of Table 12.16*b*.

The partition inequality in Theorem 12.3 is frequently referred to as *information-flow inequality*. It defines the minimal amount of information which we must have in order to compute the value of y_i for the next state. In other words, since $M[\tau(y_i)]$ is the largest partition (the least amount of information regarding the machine's state) from which we can determine the block of $\tau(y_i)$ containing the next state of the machine then, in order to compute the value of y_i for the next state, we must have at least as much information about the present state as is contained in $M[\tau(y_i)]$. Thus, knowledge of the information-flow inequalities is sufficient to specify the dependencies of the state variables and determine the direction of "information flow" in the machine.

Computing the *Mm* pairs

Having established (in Theorem 12.3) the role of *Mm* pairs in the determination of assignments with reduced dependencies, we proceed to develop a systematic procedure to generate these pairs. Let a and b be two arbitrary states of machine M, and let τ_{ab} be the partition that includes a block (ab) and leaves all other states in separate blocks. Then $m(\tau_{ab})$ is the smallest partition containing the blocks implied by the identification of (ab). Clearly, $(\tau_{ab}, m(\tau_{ab}))$ is a partition pair.

Table 12.17 Machine M_9

	NS				
PS	x_1x_2 00	01	11	10	z
A	C	A	D	B	0
B	E	C	B	D	0
C	C	D	C	E	0
D	E	A	D	B	0
E	E	D	C	E	1

Any partition τ can be expressed as a sum $\tau = \sum \tau_{ab}$, where the sum is taken over all τ_{ab} such that $\tau_{ab} \leq \tau$. In addition, since the sum of partition pairs is also a partition pair, $(\sum \tau_{ab}, \sum m(\tau_{ab}))$ must be a partition pair. Therefore, if τ is the M-partition then $(\tau, \sum m(\tau_{ab}))$ is an Mm pair.

The preceding result provides us with the basic tool for the computation of Mm pairs. First, we find the set $\{m(\tau_{ab})\}$ for all distinct a and b. This process requires $n(n-1)/2$ computations. Next, we find all possible sums of these partitions. From the preceding results, it is evident that this process generates all the m-partitions. The M-partition $\tau = M(\tau')$ corresponding to every m-partition τ' is given by $\tau = \sum \tau_{ab}$, where the sum is taken over all τ_{ab} such that $m(\tau_{ab}) \leq \tau'$. This procedure actually generates the sum of all τ_{ab} which satisfy the requirement that (τ_{ab}, τ') is a partition pair. As an example, we shall compute the Mm pairs for the machine M_9 given in Table 12.17.

First, we compute the $m(\tau_{ab})$, starting from $m(\tau_{AB})$ and continuing through all possible pairs up to $m(\tau_{DE})$. The m-partition $m(\tau_{AB})$ is found by obtaining the successors implied by the identification of A and B. From the state table we conclude that the identification of (AB) implies the identifications of (CE), (AC), and (BD). The application of the transitive rule yields

$$m(\tau_{AB}) = \{\overline{A, C, E}; \overline{B, D}\} = \tau_1'.$$

Hence, if the uncertainty regarding the present state of M, which is specified by τ_{AB}, is (AB) then the uncertainty regarding the next state of M is given by $m(\tau_{AB}) = \tau_1'$. In a similar fashion, we find the following set of distinct $m(\tau_{ab})$'s for machine M_9:

$$\begin{aligned}
m(\tau_{AC}) &= m(\tau_{DE}) = \{\overline{A, C, D}; \overline{B, E}\} = \tau_2', \\
m(\tau_{AD}) &= m(\tau_{CE}) = \{\overline{A}; \overline{B}; \overline{C, E}; \overline{D}\} = \tau_3', \\
m(\tau_{AE}) &= m(\tau_{CD}) = \pi(I), \\
m(\tau_{BC}) &= m(\tau_{BE}) = \{\overline{A}; \overline{B, C, D, E}\} = \tau_4', \\
m(\tau_{BD}) &= \{\overline{A, C}; \overline{B, D}; \overline{E}\} = \tau_5'.
\end{aligned}$$

The next step in the computation of m-partitions is to form all possible sums of the $m(\tau_{ab})$. This is accomplished by performing all pairwise sums,

then pairwise sums of the new partitions generated, and so on. In the above example, no new nontrivial m-partitions are generated in this step.

Using the above set of m-partitions, we compute next the corresponding set of M-partitions. Recalling that $M(\tau_i') = \sum \tau_{ab}$, where the sum is taken over all τ_{ab} such that $m(\tau_{ab}) \leq \tau_i'$, we obtain

$$
\begin{aligned}
M(\tau_1') &= \tau_{AB} + \tau_{AD} + \tau_{CE} + \tau_{BD} = \{\overline{A, B, D}; \overline{C, E}\} = \tau_1,\\
M(\tau_2') &= \tau_{AC} + \tau_{DE} = \{\overline{A, C}; \overline{B}; \overline{D, E}\} = \tau_2,\\
M(\tau_3') &= \tau_{AD} + \tau_{CE} = \{\overline{A, D}; \overline{B}; \overline{C, E}\} = \tau_3,\\
M(\tau_4') &= \tau_{BC} + \tau_{BE} + \tau_{AD} + \tau_{CE} = \{\overline{A, D}; \overline{B, C, E}\} = \tau_4,\\
M(\tau_5') &= \tau_{BD} = \{\overline{A}; \overline{B, D}; \overline{C}; \overline{E}\} = \tau_5.
\end{aligned}
$$

Thus, the machine M_9 possesses a set of seven Mm pairs (of which two pairs are trivial), namely,

$$
\begin{aligned}
&(\pi(I), \pi(I)),\\
(\tau_1, \tau_1') &= (\{\overline{A, B, D}; \overline{C, E}\}, \{\overline{A, C, E}; \overline{B, D}\}),\\
(\tau_2, \tau_2') &= (\{\overline{A, C}; \overline{B}; \overline{D, E}\}, \{\overline{A, C, D}; \overline{B, E}\}),\\
(\tau_3, \tau_3') &= (\{\overline{A, D}; \overline{B}; \overline{C, E}\}, \{\overline{A}; \overline{B}; \overline{C, E}; \overline{D}\}),\\
(\tau_4, \tau_4') &= (\{\overline{A, D}; \overline{B, C, E}\}, \{\overline{A}; \overline{B, C, D, E}\}),\\
(\tau_5, \tau_5') &= (\{\overline{A}; \overline{B, D}; \overline{C}; \overline{E}\}, \{\overline{A, C}; \overline{B, D}; \overline{E}\}),\\
&(\pi(0), \pi(0)).
\end{aligned}
$$

The Mm-lattice can now be drawn in a straightforward manner.

The above Mm pairs characterize the machine and contain all the information regarding its structure. In addition to numerous partition pairs that can be generated from these Mm pairs, two closed partitions π_1 and π_2 exist, where

$$
\begin{aligned}
\pi_1 &= \{\overline{A, D}; \overline{B}; \overline{C, E}\},\\
\pi_2 &= \{\overline{A}; \overline{B}; \overline{C, E}; \overline{D}\}.
\end{aligned}
$$

The closed partitions are generated by enlarging the m-partition and refining the M-partition of the Mm pair (τ_3, τ_3').

State assignments based on partition pairs

We shall now apply the principles developed in this section, and our knowledge about the information flow in the machine M_9, to obtain an assignment in which the dependencies of the variables will be reduced. For the example at hand our aim is to obtain a three-variable assignment. Consequently, we are seeking three partitions, $\lambda_1, \lambda_2, \lambda_3$, of two blocks each, such that

$$\lambda_1 \cdot \lambda_2 \cdot \lambda_3 = \pi(0).$$

For each λ_i, we shall determine the corresponding $M(\lambda_i)$ and thus obtain three partition pairs, $(M(\lambda_1), \lambda_1), (M(\lambda_2), \lambda_2), (M(\lambda_3), \lambda_3)$, from which the structure of the machine can be determined.

To each partition λ_i we assign one state variable, y_i (in general, there are $\lceil \log_2 \#(\lambda_i) \rceil$ state variables). Then $M(\lambda_i)$ is the partition containing the smallest amount of information from which we can compute the value of y_i assigned to the block of λ_i that contains the next state of the machine. From Theorem 12.3 it is evident that a reduction in the dependency of the variable assigned to a partition λ_i is achieved if $M(\lambda_i)$ is greater than or equal to the product of a small subset of partitions, $\lambda_1, \lambda_2, \lambda_3$. The variables assigned to the partitions in the subset provide y_i with at least that information specified by $M(\lambda_i)$.

In order to select the partitions $\lambda_1, \lambda_2, \lambda_3$, we look for two-block partitions in the set of m-partitions τ_i's. In particular, if a variable y_i assigned to λ_i is to depend on just one other variable assigned to the blocks of λ_j then $\lambda_j \leq M(\lambda_i)$ and $M(\lambda_i)$ can have at most two blocks. Thus, as our initial selection, let $\lambda_1 = \tau_1'$. Since $M(\tau_1')$ consists of two blocks, we may select it as the second partition, i.e., $\lambda_2 = M(\tau_1')$. Hence the variable Y_1 defined by λ_1 will depend only on the information provided by y_2, which is defined by λ_2. As λ_1 and λ_2 have already been selected, the selection of λ_3 is simple, since it must satisfy $\lambda_1 \cdot \lambda_2 \cdot \lambda_3 = \pi(0)$. We thus choose $\lambda_3 = \tau_2'$. The partitions $\lambda_1, \lambda_2, \lambda_3$ and their corresponding M-partitions $M(\lambda_1), M(\lambda_2), M(\lambda_3)$ are given as follows:

$$
\begin{aligned}
(M(\lambda_1), \lambda_1) &= (\{\overline{A, B, D}; \overline{C, E}\}, \{\overline{A, C, E}; \overline{B, D}\}),\\
(M(\lambda_2), \lambda_2) &= (\{\overline{A, D}; \overline{B}; \overline{C, E}\}, \{\overline{A, B, D}; \overline{C, E}\}),\\
(M(\lambda_3), \lambda_3) &= (\{\overline{A, C}; \overline{B}; \overline{D, E}\}, \{\overline{A, C, D}; \overline{B, E}\}).
\end{aligned}
$$

Note that λ_2 is not an m-partition but, since $\lambda_2 > \tau_3'$, we have $M(\lambda_2) \geq M(\tau_3')$.

From the way in which we selected the above partition pairs it is evident that Y_1 depends only on y_2, since λ_2 provides all the information that Y_1 requires as specified by $M(\lambda_1)$. In order to determine the dependencies of Y_2 and Y_3, we check to see whether there exists a partition $\lambda_i \leq M(\lambda_2)$ or $\lambda_j \leq M(\lambda_3)$. Since there are no such partitions, the next step is to check whether we can form a product of two partitions such that $\lambda_i \cdot \lambda_j \leq M(\lambda_2)$ or $\lambda_p \cdot \lambda_q \leq M(\lambda_3)$. Indeed, this can be accomplished, since

$$
\begin{aligned}
\lambda_2 \cdot \lambda_3 &< M(\lambda_2),\\
\lambda_1 \cdot \lambda_3 &< M(\lambda_3).
\end{aligned}
$$

Consequently, Y_2 depends on the information supplied by y_2 and y_3, while Y_3 receives its inputs from y_1 and y_3. The functional dependencies of the next-state variables are summarized as follows:

$$
\begin{aligned}
Y_1 &= f_1(x_1, x_2, y_2),\\
Y_2 &= f_2(x_1, x_2, y_2, y_3),\\
Y_3 &= f_3(x_1, x_2, y_1, y_3).
\end{aligned}
$$

The schematic diagram of the circuit structure is shown in Fig. 12.13.

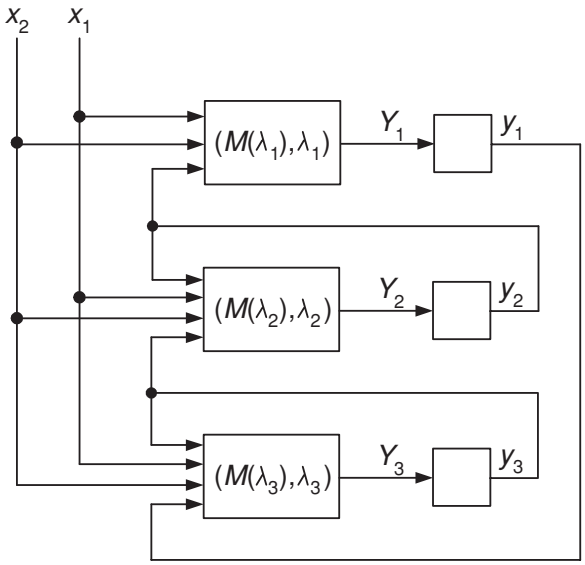


Fig. 12.13 Schematic diagram of the structure of M_9 when realized using λ_1, λ_2, and λ_3.

12.8 Decomposition

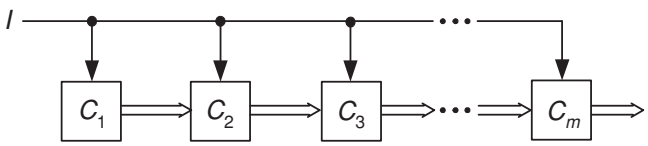
In the preceding sections we have studied the relationship between the state-assignment problem and the structure of sequential machines and have determined necessary and sufficient conditions for a machine to be decomposable. Our objective in this section is to investigate further the properties of decomposable machines and of various component machines.

Serial decomposition

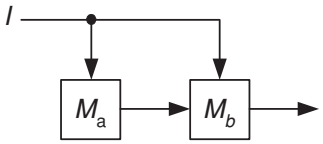
We shall first determine the conditions for a machine M to be decomposable into a serial (cascade) chain of component machines $C_1, C_2, \ldots, C_m$ in which the outputs of any component may be used as inputs to other components. If an output of machine C_i is an input of machine C_j then C_i is said to be a *predecessor* of C_j and C_j is said to be a *successor* of C_i. We shall assume that the component machines *operate concurrently*, i.e., that the next state of each component depends on its present state, on the current values of external inputs, and on the present state of its predecessors. We shall assume further that the component machines form a *loop-free interconnection*; i.e., if C_i or any of its successors or successors of successors, etc., is a predecessor of C_j then C_j must not be a predecessor of C_i. A schematic diagram of such a serial decomposition is shown in Fig. 12.14*a*.

Theorem 12.4 *Let a machine M be realizable as a serial loop-free connection of m components $C_1, C_2, \ldots, C_m$; then there exists a set of m closed partitions $\{\pi_1, \pi_2, \ldots, \pi_m\}$ such that $\pi_1 \geq \pi_2 \geq \cdots \geq \pi_m$ and $\pi_m = \pi(0)$. Conversely,*

Fig. 12.14 Serial decomposition of a machine.



(*a*) Cascaded chain. (The double arrows indicate a transfer of information from all predecessor stages.)



(*b*) Block diagram of the cascaded chain.

such a set of closed partitions is a sufficient condition for the existence of a serial decomposition in which C_i is a predecessor of C_j if and only if $\pi_i \geq \pi_j$.

Proof Suppose that the machine M has been realized as a serial connection of m components, as shown in Fig. 12.14*a*. For the purpose of analysis we may divide these components into two groups, as shown in Fig. 12.14*b*. The first group, denoted M_a, consists of k components and the second group, denoted M_b, consists of $m - k$ components. If we let k equal 1 then, by Theorem 12.1, there exists a closed partition π_1 on the states of M. Similarly, if we group the machines together as (C_1, C_2) and $(C_3, C_4, \ldots, C_m)$, we obtain another serial decomposition, of the type shown in Fig. 12.14*b*, to which there corresponds another closed partition π_2 on the states of the machine M.

To determine the relation between π_1 and π_2, note that, since C_1 distinguishes the blocks of π_1, each block of π_1 in fact corresponds to a state of C_1. Similarly, each block of π_2 corresponds to a state of the composite machine (C_1, C_2). However, since (C_1, C_2) can be decomposed into C_1 in series with C_2, it follows that each state of C_1 represents one or more states of the composite machine (C_1, C_2). Consequently, each block of π_1 contains one or more blocks of π_2, i.e., $\pi_1 \geq \pi_2$. There exist m possible ways (one of which is trivial) of arranging the component machines in two groups, $(C_1, \ldots, C_k)$ and $(C_{k+1}, \ldots, C_m)$. Hence, there exist m closed partitions $\pi_1 \geq \pi_2 \geq \cdots \geq \pi_m$. Note that the equality sign in the above relation can be omitted, since it corresponds to a degenerate case. In fact, if $\pi_{k-1} = \pi_k$ then the component C_k is redundant and may be deleted.

The converse can be proved by illustrating the construction of the decomposed machine. Let $\pi_1 > \pi_2 > \cdots > \pi_m$ be a set of closed partitions on M. Select another set of partitions, $\tau_1, \tau_2, \ldots, \tau_{m-1}$, such that, for each value of i in the range $1 \leq i \leq m - 1$,

$$\pi_i \cdot \tau_i = \pi_{i+1}$$

and

$$\pi_1 \cdot \tau_1 \cdot \tau_2 \cdot \quad \ldots \quad \cdot \tau_{m-1} = \pi(0).$$

The first component, C_1, contains $\lceil \log_2 \#(\pi_1) \rceil$ state variables, which are assigned to distinguish the blocks of π_1. Thus, C_1 is independent of the remaining components. The second component, C_2, consists of the $\lceil \log_2 \#(\tau_1) \rceil$ variables assigned to the blocks of τ_1. Since τ_1 is not necessarily closed, C_2 depends on C_1. However, since $\pi_1 \cdot \tau_1 = \pi_2$, C_2 is independent of the remaining components $C_3, \ldots, C_m$. In a similar manner, the decomposed machine is constructed in such a way that each component C_k is independent of $C_{k+1}, \ldots, C_m$ and is a function of $C_1, \ldots, C_k$. ◇

Theorem 12.4 establishes the concept of *information flow* in a sequential machine, i.e., a machine realized as a serial connection of smaller components. In fact we have proved that, in the cascaded chain, information flows from component C_i to C_j if and only if $\pi_i \geq \pi_j$.

Example Consider the machine M_{10} given in Table 12.18. It has three closed partitions (including the zero partition) and an output-consistent partition λ_o. Since $\pi_a > \pi_b > \pi_0$, M_{10} is decomposable into three components connected in series such that each component is a two-state machine:

Table 12.18 Machine M_{10}

	NS		
PS	$x=0$	$x=1$	z
A	G	D	1
B	H	C	0
C	F	G	1
D	E	G	0
E	C	B	1
F	C	A	0
G	A	E	1
H	B	F	0

$$\pi_0 = \pi(0),$$
$$\pi_a = \{\overline{A, B, G, H}; \overline{C, D, E, F}\},$$
$$\pi_b = \{\overline{A, B}; \overline{C, D}; \overline{E, F}; \overline{G, H}\},$$
$$\lambda_o = \{\overline{A, C, E, G}; \overline{B, D, F, H}\}.$$

The machine C_a, which is derived from π_a, consists of $\#(\pi_a) = 2$ states and, therefore, can be realized by a single state variable, y_a. The second component, C_b, is derived from a partition τ_1 such that $\pi_a \cdot \tau_1 = \pi_b$. One

possible such partition, τ_1, is given by

$$\tau_1 = \{\overline{A, B, C, D}; \overline{E, F, G, H}\}.$$

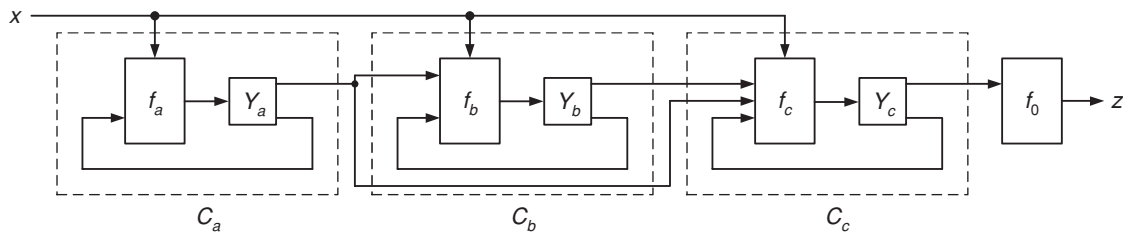
Since $\#(\tau_1) = 2$, the machine C_b will consist of a single variable, y_b. Variables y_a and y_b are actually assigned to the blocks of the closed partition π_b and, therefore, are independent of the remaining variable, which is assigned to the blocks of some partition τ_2 such that $\tau_2 \cdot \pi_b = \pi(0)$. Several partitions satisfy the last requirement. It is desirable, however, to select (whenever possible) a partition yielding simpler output circuits, i.e., for which $\tau_2 \leq \lambda_o$. A choice satisfying this condition is

$$\tau_2 = \lambda_o = \{\overline{A, C, E, G}; \overline{B, D, F, H}\}.$$

An assignment based on the above partitions will yield the following functional relationships:

$$\begin{aligned} Y_a &= f_a(x, y_a), \\ Y_b &= f_b(x, y_a, y_b), \\ Y_c &= f_c(x, y_a, y_b, y_c), \\ z &= f_0(y_c). \end{aligned}$$

The schematic diagram of this realization and the π-lattice of M_{10} are shown in Fig. 12.15.



(*a*) Serial decomposition.

(*b*) π-lattice.

Fig. 12.15 Schematic diagram and π-lattice of M_{10}.

The machine M_{10} has thus been decomposed into three components connected in series. It is often necessary to determine the state table of each of these components, a task accomplished as follows. The state diagram of C_a is obtained by constructing the implication graph of π_a. It consists of two vertices,

Table 12.19 State tables of the component machines realizing M_{10}

	x		
PS	0	1	y_a
P	P	Q	0
Q	Q	P	1

(*a*) C_a

	$y_a x$				
PS	00	01	10	11	y_b
α	β	α	β	β	0
β	α	β	α	α	1

(*b*) C_b

PS	i_1	i_2	y_b
α	β	α	0
β	α	β	1

(*c*) C_b – reduced form

	$y_a y_b x$								
PS	000	001	010	011	100	101	110	111	z
γ	γ	δ	γ	γ	δ	γ	γ	δ	1
δ	δ	γ	δ	δ	γ	γ	γ	γ	0

(*d*) C_c

PS	I_1	I_2	I_3	z
γ	γ	δ	γ	1
δ	δ	γ	γ	0

(*e*) C_c – reduced form

P and Q, corresponding respectively to the blocks $(ABGH)$ and $(CDEF)$. The state table of C_a, which is identical to the implication table derived from π_a, is given in Table 12.19*a*. The output of C_a is associated with its state and is identical to the value of y_a.

The inputs to C_b are x and y_a, and its state-dependent output is y_b. It contains two states, α and β, corresponding respectively to the blocks $(ABCD)$ and $(EFGH)$ of τ_1. The state table of C_b is shown in Table 12.19*b*; the input symbol 00 means that C_a is in state P, i.e., $y_a = 0$, and that the external input value is $x = 0$. When C_a is in state P and C_b is in state α then M_{10} is in either state A or state B. From these states C_b goes to state β, which corresponds to G and H. When C_a is in state P, C_b is in state β, and the input value $x = 0$ is applied, C_b is to go to state α, which corresponds to states A and B in M_{10}. The entire table is completed in a similar fashion. The composite states of C_a and C_b correspond to the blocks of π_b. Since $\pi_a = \{P; Q\}$ and $\tau_1 = \{\alpha; \beta\}$,

$$\pi_b = \pi_a \cdot \tau_1 = \{P\alpha; P\beta; Q\alpha; Q\beta\} = \{\overline{A, B}; \overline{G, H}; \overline{C, D}; \overline{E, F}\}.$$

Finally, we note that C_b can be reduced to a machine with only two input symbols, since the next-state entries in three columns of C_b are identical. If we define i_1 and i_2 as

$$i_1 = x' + y_a,$$
$$i_2 = xy_a'$$

we obtain the reduced form of C_b, as shown in Table 12.19*c*.

The machine C_c consists of two states, γ and δ, corresponding to the blocks of $\tau_2 = \{\overline{A, C, E, G}; \overline{B, D, F, H}\} = \{\gamma; \delta\}$, as shown in Table 12.19*d*. It receives three inputs, x, y_a, and y_b, and produces one output, z. The input symbol 000 in this table means that C_a and C_b are in states P and α, respectively, and

Table 12.20 Machine M_{11}

	NS		
PS	$x = 0$	$x = 1$	z
A	D	G	0
B	C	E	0
C	H	F	0
D	F	F	0
E	B	B	0
F	G	D	0
G	A	B	0
H	E	C	1

$x = 0$. This, in turn, implies that M_{10} is in either state A or B, depending on whether C_c is in state γ or δ, respectively. The 0-successors of A and B are G and H, which correspond to $P\beta\gamma$ and $P\beta\delta$, respectively. Therefore, the entries in column 000 of C_c are γ and δ. In a similar fashion, the state table of C_c is derived from Table 12.18 and set of partitions π_a, τ_1, and τ_2. By making the appropriate input assignment, Table 12.19*d* may be reduced to the form shown in Table 12.19*e*.

Parallel decompositions

We have already shown that a necessary and sufficient condition for a sequential machine M to be decomposable into two independent components operating in parallel is the existence of two closed partitions (or covers), π_1 and π_2, such that $\pi_1 \cdot \pi_2 = \pi(0)$. This result can be easily generalized to a decomposition into m parallel components, which can be accomplished if and only if there exists a set of m closed partitions (or covers) on M such that $\pi_1 \cdot \pi_2 \cdot \ldots \cdot \pi_m = \pi(0)$.

The machine M_{11} given in Table 12.20 has the π-lattice of Fig. 12.16*a* and the following nontrivial closed partitions:

$$\pi_a = \{\overline{A, B}; \overline{C, D}; \overline{E, G}; \overline{F, H}\},$$
$$\pi_b = \{\overline{A, H}; \overline{B, F}; \overline{C, G}; \overline{D, E}\},$$
$$\pi_c = \{\overline{A, B, F, H}; \overline{C, D, E, G}\}.$$

Since $\pi_a \cdot \pi_b = \pi(0)$, a parallel decomposition of M_{11} is possible. However, $\lceil \log_2 \#(\pi_a) \rceil + \lceil \log_2 \#(\pi_b) \rceil = 4$ and so such a decomposition requires four state variables. The state tables of the component machines M_a and M_b, which correspond respectively to π_a and π_b, are given in Table 12.21. The schematic diagram of the realization is shown in Fig. 12.16*b*.

Since this realization requires four state variables, we next seek another decomposition, one which will require only three variables. Ultimately, our

Table 12.21 Parallel decomposition of M_{11}

		NS		
PS		$x=0$	$x=1$	z_1
A, B	a	b	c	0
C, D	b	d	d	0
E, G	c	a	a	0
F, H	d	c	b	1

(a) M_a

		NS		
PS		$x=0$	$x=1$	z_2
A, H	α	δ	γ	1
B, F	β	γ	δ	0
C, G	γ	α	β	0
D, E	δ	β	β	0

(b) M_b

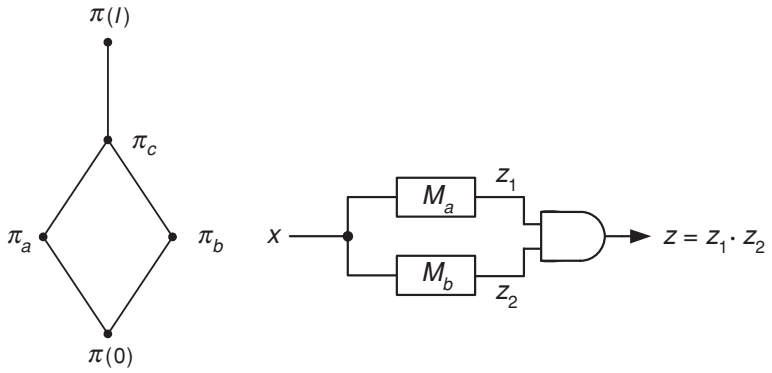


Fig. 12.16 Parallel decomposition of M_{11}.

(a) π-lattice. (b) Schematic diagram.

aim is to determine whether the machines M_a and M_b can each be serially decomposed in such a manner that both have an identical independent component. If such a component can be found, it may be "factored out" to serve as a common predecessor for both M_a and M_b. A necessary condition for the existence of such a common component is that both M_a and M_b can be serially decomposed; i.e., that both M_a and M_b have nontrivial closed partitions on their respective states. Clearly, the largest component machine that can be factored out is given by the smallest closed partition that is greater than π_a and π_b, i.e., lub $\pi_a + \pi_b$. For the machine M_{11},

$$\pi_c = \pi_a + \pi_b = \{\overline{A, B, F, H}; \overline{C, D, E, G}\}.$$

Since lub π_c is nontrivial, a two-state component can be factored out and thus a decomposition of the form shown in Fig. 12.17 is possible for M_{11}. The common factor M_c in series with M_d realizes M_a, while M_c in series with M_e realizes M_b. The factor M_c and the components M_d and M_e are given in Table 12.22.

Table 12.22 The component machines corresponding to Fig. 12.17

		x		
PS		0	1	y_c
A, B, F, H	P	Q	Q	0
C, D, E, G	Q	P	P	1

(*a*) M_c

		$y_c x$				
PS		00	01	10	11	z_d
A, B, C, D	r	r	s	s	s	0
E, F, G, H	s	s	r	r	r	1

(*b*) M_d

		$y_c x$				
PS		00	01	10	11	z_d
A, C, G, H	u	v	u	u	v	1
B, D, E, F	v	u	v	v	v	0

(*c*) M_e

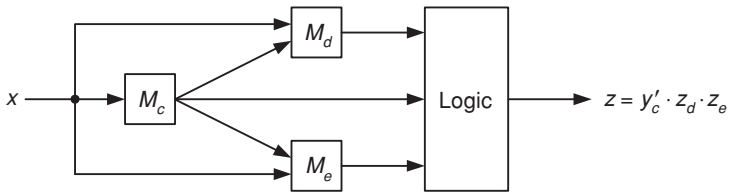


Fig. 12.17 Another decomposition of M_{11}.

Decompositions with specified components

We have studied several machine structures and determined the conditions for a machine to be decomposable into these structures. Our present objective is to determine whether a machine can be decomposed in such a manner that one (or more) of its components is specified. One possible approach to the solution of this problem is to check all closed partitions and covers and determine whether any of them yields the desired specified component. This approach, however, is long and impractical, and so a new technique to handle this type of decompositions will be developed.

As an example, consider the machines M_{12} and C_1 given in Tables 12.23 and 12.24, respectively. Our objective is to determine whether M_{12} can be serially

Table 12.23 Machine M_{12}

	NS		z	
PS	I_1	I_2	I_1	I_2
A	C	D	0	0
B	D	E	0	1
C	A	C	0	0
D	B	D	0	0
E	F	E	1	1
F	C	D	1	1

Table 12.24 Machine C_1

	NS	
PS	I_1	I_2
P	S	Q
Q	R	Q
R	S	Q
S	P	S

Table 12.25 Composite machine for M_{12} and C_1 and initial states A and P

	NS	
PS	I_1	I_2
AP	CS	DQ
CS	AP	CS
DQ	BR	DQ
BR	DS	EQ
DS	BP	DS
EQ	FR	EQ
BP	DS	EQ
FR	CS	DQ

decomposed in such a way that C_1 is the predecessor component. In order to determine whether such a decomposition is possible, it is necessary to establish what information regarding the states of M_{12} is contained in C_1. This can be accomplished by constructing a composite machine that contains both M_{12} and C_1 and is defined as follows.

Let the *general composite machine*, corresponding to the two machines M_1 and M_2, having sets of states R and S, respectively, be the machine that contains the set of states $R \times S$. We shall use $R_i S_j$ to denote the state of the general composite machine which corresponds to R_i in M_1 and (simultaneously) S_j in M_2. For two machines M_1 and M_2 having simultaneous initial states R_1 and S_1, the *composite machine* is that having initial state $R_1 S_1$ and subsequent states implied in chain fashion by $R_1 S_1$ and its successors.

The composite machine corresponding to the machines M_{12} and C_1 and to the initial states A and P respectively is given in Table 12.25. Starting with AP, the application of the input symbol I_1 takes M_{12} to state C and C_1 to state S. Therefore, the I_1-successor of AP is CS. In a similar fashion, we conclude that the I_2-successor of AP is DQ, and so on. Next, we obtain the successors of states CS and DQ, and this process continues until no new states are generated.

In general, if M_1 has n_1 states and M_2 has n_2 states, the general composite machine has $n_1 \cdot n_2$ states. However, it may have as many as $n_1 \cdot n_2$ states, or as few as the smaller of n_1 or n_2 states. The I_k-successor of state $R_i S_j$ of the composite machine is obtained from the I_k-successors of R_i and S_j in their respective machines, i.e., if $I_k R_i$ is R_p and $I_k S_j$ is S_q then the I_k-successor of $R_i S_j$ is $R_p S_q$.

For the machine M_{12} to be serially decomposable in such a way that C_1 is the predecessor component, it is necessary that M_{12} should have a closed cover whose corresponding implication graph is *equivalent* to the state diagram of C_1; i.e., both graphs must be isomorphic and the labels of the arcs connecting corresponding vertices must be identical. This closed cover can be obtained from the composite machine of Table 12.25 in a straightforward manner. From the names of the new states in this table, it can be concluded that when C_1 is in state P the composite machine can be in either state AP or state BP, and M_{12} can only be in A or B. Similarly, when C_1 is in state S, M_{12} can only be in state C or D, and so on. We can thus form a cover φ on the states of M_{12} such that two states (say R_i and R_j) are in the same block of φ if and only if they are associated with the same state of C_1 (say S_k); i.e., the composite machine of M_{12} and C_1 contains the states $R_i S_k$ and $R_j S_k$. Thus, for machine M_{12}, we have

$$\varphi = \{\overline{A, B}; \overline{D, E}; \overline{B, F}; \overline{C, D}\}.$$

Blocks (A, B) and (D, E) of φ correspond respectively to states P and Q in C_1, while (B, F) and (C, D) correspond respectively to states R and S. Consequently, knowledge of the state of C_1 is always sufficient to obtain the state of M_{12} to within at most two states.

In order to complete the synthesis it is necessary to specify the successor component. A simple way to accomplish this is first to split states B and D of machine M_{12} in such a way that $\pi = \{\overline{A, B'}; \overline{D', E}; \overline{B'', F}; \overline{C, D''}\}$ is a closed partition on the states of the augmented machine. The predecessor component of the augmented machine is isomorphic to C_1, while the successor component, which consists of two states, distinguishes the blocks of a partition τ given by

$$\tau \cdot \{\overline{A, B'}; \overline{D', E}; \overline{B'', F}; \overline{C, D''}\} = \pi(0).$$

One possibility is

$$\tau = \{\overline{A, D', D'', F}; \overline{B', B'', E, C}\}.$$

The state tables of the augmented machine and the successor component are obtained in the usual manner, as illustrated in the previous sections.

*12.9 Synthesis of multiple machines

We shall now generalize the decomposition problem to include the simultaneous decomposition of two or more machines. More precisely, given two reduced

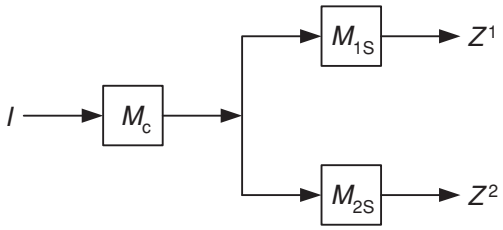


Fig. 12.18 Two machines having a common predecessor component M_C.

Table 12.26 Two machines, M_1 and M_2, to be decomposed simultaneously

	NS		
R	I_1	I_2	Z^1
R_1	R_1	R_2	Z_1^1
R_2	R_2	R_3	Z_2^1
R_3	R_3	R_4	Z_3^1
R_4	R_4	R_1	Z_4^1

(a) M_1

	NS		
S	I_1	I_2	Z^2
S_1	S_3	S_2	Z_1^2
S_2	S_4	S_3	Z_2^2
S_3	S_1	S_4	Z_3^2
S_4	S_2	S_1	Z_4^2

(b) M_2

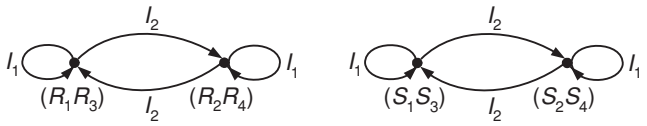


Fig. 12.19 Implication graphs.

(a) M_1. (b) M_2.

machines M_1 and M_2 having the same input alphabet I, which are initially in states R_1 and S_1 respectively, we wish to find three machines M_C, M_{1S}, and M_{2S}, where M_C is a common predecessor component whose output feeds the successors M_{1S} and M_{2S} in such a way that M_C and M_{1S} form a serial decomposition of M_1 while M_C and M_{2S} form a serial decomposition of M_2. Figure 12.18 shows the desired structure, in which Z^1 and Z^2 are the outputs of M_{1S} and M_{2S}, respectively. When a maximum common predecessor component exists, the total state variables required for the realization is minimum, while the total output logic circuitry is not more complex than if the two machines were realized separately.

The common predecessor machine

As an example, consider the two reduced Moore-type machines given in Table 12.26. The implication graphs of machines M_1 and M_2, for the initial identifications of (R_1R_3) and (S_1S_3) respectively, are shown in Fig. 12.19. These closed

Table 12.27 Machine M_C

	NS	
PS	I_1	I_2
P	P	Q
Q	Q	P

graphs are equivalent since they are isomorphic and the labels of arcs that connect corresponding vertices are identical. We have already established that the closed implication graph of a sequential machine M is actually equivalent to the state diagram of the predecessor component in a serial decomposition of M. Consequently, each graph in Fig. 12.19 can serve as a state diagram of the predecessor component in the serial decomposition of the respective machine. In addition, since the two graphs are equivalent they correspond to equivalent machines. Because the two predecessor components are equivalent, one may be removed and the other retained as the common predecessor component.

The graphs of Fig. 12.19 correspond respectively to the closed partitions

$$\pi_1 = \{\overline{R_1, R_3}; \overline{R_2, R_4}\} \qquad \text{and} \qquad \pi_2 = \{\overline{S_1, S_3}; \overline{S_2, S_4}\}.$$

If we denote the first and second blocks of each partition by P and Q respectively then we obtain the implication table in Table 12.27. This is the state table of the common predecessor component M_C. Successor components M_{1S} and M_{2S} can be obtained by using the methods developed in the foregoing section.

From the preceding example, it is evident that *a collection of two (or more) machines contains a common predecessor component M_C if and only if they possess equivalent implication graphs; the vertices and arcs of this common graph are in one-to-one correspondence with the states and state transitions respectively of M_C*. The procedure for finding the equivalent graphs is not entirely systematic, however, since it depends on the selection of the initial state identifications. This limitation can be overcome by using the composite machine, as is shown subsequently.

The composite machine corresponding to M_1 and M_2 and to initial states R_1 and S_1 is given in Table 12.28. It consists of eight states. While the composite machine includes all states of M_1 and M_2, it does not include all combinations of these states; e.g., R_1S_2 is not encountered when any of the eight states of the composite machine is selected as the initial state. Furthermore, if M_1 is initially in state R_1, then M_2 can be started only in either S_1 or S_3, since the only combinations of states included in the composite machine are R_1S_1 and R_1S_3. Thus, the choice of an initial state, in effect, locks the two machines together, in an operational sense.

Table 12.28 Composite machine for M_1 and M_2 and initial states R_1 and S_1

	NS		
PS	I_1	I_2	Z^1Z^2
R_1S_1	R_1S_3	R_2S_2	$Z_1^1Z_1^2$
R_1S_3	R_1S_1	R_2S_4	$Z_1^1Z_3^2$
R_2S_2	R_2S_4	R_3S_3	$Z_2^1Z_2^2$
R_2S_4	R_2S_2	R_3S_1	$Z_2^1Z_4^2$
R_3S_3	R_3S_1	R_4S_4	$Z_3^1Z_3^2$
R_3S_1	R_3S_3	R_4S_2	$Z_3^1Z_1^2$
R_4S_4	R_4S_2	R_1S_1	$Z_4^1Z_4^2$
R_4S_2	R_4S_4	R_1S_3	$Z_4^1Z_2^2$

Using the above procedure, we have transformed the two-machine problem into the well-known single-machine decomposition problem. The methods developed in the preceding sections are now applicable to the composite machine which contains the two machines M_1 and M_2.

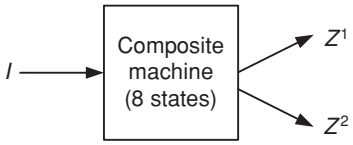
Decomposing the composite machine

Let us now define two partitions, π_R and π_S, on the states of the composite machine such that two states are placed in the same block of π_R if and only if their labels start with the same state R_i in M_1; two states are placed in the same block of π_S if and only if their names end with the same state S_j in M_2. Such partitions are often referred to as *state-consistent* partitions and are derived directly from the composite machine.

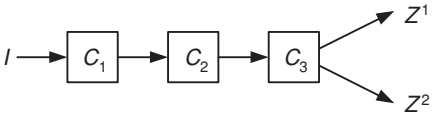
Example The state-consistent partitions for the composite machine of Table 12.28 are

$$\pi_R = \{\overline{R_1S_1,\ R_1S_3};\ \overline{R_2S_2,\ R_2S_4};\ \overline{R_3S_3,\ R_3S_1};\ \overline{R_4S_4,\ R_4S_2}\},$$
$$\pi_S = \{\overline{R_1S_1,\ R_3S_1};\ \overline{R_2S_2,\ R_4S_2};\ \overline{R_1S_3,\ R_3S_3};\ \overline{R_2S_4,\ R_4S_4}\}.$$

The block (R_1S_1, R_1S_3) of π_R corresponds to state R_1 in M_1, the block $(R_1S_1,\ R_3S_1)$ of π_S corresponds to state S_1 in M_2, and so on. From the way in which the state-consistent partitions π_R and π_S are constructed, it is evident that they correspond to the zero partitions on the set of states of the machines M_1 and M_2, respectively. Consequently, the implication graphs corresponding to π_R and π_S are equivalent to the state graphs of M_1 and M_2 respectively; therefore these partitions are closed with respect to the states of the composite machine.



(*a*) Simple realization.



(*b*) Decomposition of the composite machine.

Fig. 12.20 Two possible realizations of the composite machine.

From the composite machine of Table 12.28, it is apparent that the required outputs Z^1 and Z^2 can be generated by a machine having three state variables and the appropriate output logic rather than by two separate machines having a total of four state variables. This result is illustrated in Fig. 12.20*a*. We also observe that $\pi_R \cdot \pi_S = \pi(0)$, which, since both partitions are closed, is the condition for a parallel decomposition of the composite machine. In this case of course the result is simply the original two machines, M_1 and M_2, realized separately and having four state variables.

The composite machine is next examined for other possible decompositions, following the techniques previously developed. For example, the partitions

$$\pi_1 = \{\overline{R_1S_1,\ R_2S_2,\ R_3S_3,\ R_4S_4};\ \overline{R_1S_3,\ R_2S_4,\ R_3S_1,\ R_4S_2}\}$$

and

$$\pi_2 = \{\overline{R_1S_1,\ R_3S_3};\ \overline{R_2S_2,\ R_4S_4};\ \overline{R_1S_3,\ R_3S_1};\ \overline{R_2S_4,\ R_4S_2}\}$$

are easily shown to be closed and, since $\pi_1 > \pi_2$, a cascade realization of the type shown in Fig. 12.20*b* results, where each component, C_1, C_2, and C_3, is a two-state machine.

At this point, we turn our attention to the question of determining whether a common predecessor component exists for M_1 and M_2 and, if several such components exist, how to find the largest. From the results of the preceding section and from the properties of the composite machine and the state-consistent partitions π_R and π_S, it is evident that a common component exists if and only if we can find a closed partition π_C such that $\pi_C > \pi_R$ and $\pi_C > \pi_S$. Clearly, the smallest partition that satisfies these inequalities and, thus, yields the largest common component M_C, is

$$\pi_C = \pi_R + \pi_S.$$

For our example, we obtain

$$\pi_C = \pi_R + \pi_S = \{\overline{R_1S_1,\ R_1S_3,\ R_3S_1,\ R_3S_3};\ \overline{R_2S_2,\ R_2S_4,\ R_4S_2,\ R_4S_4}\}.$$

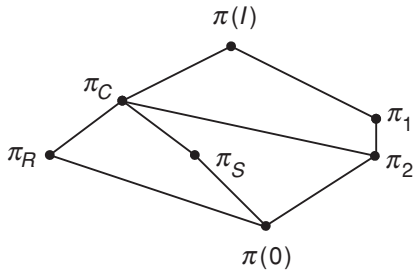


Fig. 12.21 The π-lattice for the composite machine.

Thus, a common predecessor component consisting of one state variable exists. The resulting decomposition is shown in Fig. 12.18. It is easy to verify that this machine is identical to that obtained using the implication graphs (see Table 12.27). Successor machines M_{1S} and M_{2S} (each consisting of one state variable) are obtained by partitions τ_{1S} and τ_{2S}, respectively, such that

$$\pi_C \cdot \tau_{1S} = \pi_R \qquad \text{and} \qquad \pi_C \cdot \tau_{2S} = \pi_S.$$

Possible partitions are

$$\tau_{1S} = \{\overline{R_1S_1,\ R_1S_3,\ R_2S_2,\ R_2S_4};\ \overline{R_3S_1,\ R_3S_3,\ R_4S_2,\ R_4S_4}\},$$
$$\tau_{2S} = \{\overline{R_1S_1,\ R_3S_1,\ R_2S_2,\ R_4S_2};\ \overline{R_1S_3,\ R_3S_3,\ R_2S_4,\ R_4S_4}\}.$$

Clearly, Z^1 and Z^2 are each dependent upon only two state variables and the entire machine requires a total of three state variables.

The lattice of all closed partitions on the set of states of the composite machine is shown in Fig. 12.21. However, it is of interest that our two-machine cascade decomposition has been obtained without searching for closed partitions; π_R and π_S were obtained directly by inspection of the composite machine while π_C followed from the addition of the two partitions π_R and π_S. Thus, the process involves a minimum of computation or manipulation.

Notes and references

The structure theory of machines and the study of machine decomposition were originated by Hartmanis [5] in 1960 and further developed in a series of papers by Hartmanis [6], Stearns and Hartmanis [14], Karp [8], Yoeli [15, 16], and Kohavi [9, 10]. The concept of closed covers and the procedure for augmenting a machine by state splitting were introduced by Kohavi [9] and further developed to cover multiple machines by Kohavi and Smith [11] and Smith and Kohavi [13]. Other contributions to general machine-structure theory include Krohn and Rhodes [12], Zeiger [17], and Gill [4]. A comprehensive treatment of structure and decomposition theory can be found in the book by Hartmanis and Stearns [7].

The state-assignment problem has been treated from different points of views by many authors. Of particular interest are the papers by Armstrong [1, 2] and Dolotta and McCluskey [3].

[1] Armstrong, D. B.: "A programmed algorithm for assigning internal codes to sequential machines," *IRE Trans. Electron. Computers*, vol. EC-11, no. 4, pp. 466–472, August 1962.
[2] Armstrong, D. B.: "On the efficient assignment of internal codes to sequential machines," *IRE Trans. Electron. Computers*, vol. EC-11, no. 5, pp. 611–622, October 1962.
[3] Dolotta, T. A., and E. J. McCluskey, Jr: "The coding of internal states of sequential circuits," *IEEE Trans. Electron. Computers*, vol. EC-13, no. 5, pp. 549–562, October 1964.
[4] Gill, A.: "Cascaded finite-state machines," *IRE Trans. Electron. Computers*, vol. EC-10, no. 3, pp. 366–370, September 1961.
[5] Hartmanis, J.: "Symbolic analysis of a decomposition of information processing machines," *Information and Control*, vol. 3, no. 2, pp. 154–178, June 1960.
[6] Hartmanis, J.: "On the state assignment problem for sequential machines I," *IRE Trans. Electron. Computers*, vol. EC-10, pp. 157–165, June 1961.
[7] Hartmanis, J., and R. E. Stearns: *Algebraic Structure Theory of Sequential Machines*, Prentice-Hall, Englewood Cliffs NJ, 1966.
[8] Karp, R. M.: "Some techniques of state assignment for synchronous sequential machines," *IEEE Trans. Electron. Computers*, vol. EC-13, no. 5, pp. 507–518, October 1964.
[9] Kohavi, Z.: "Secondary state assignment for sequential machines," *IEEE Trans. Electron. Computers*, vol. EC-13, no. 3, pp. 193–203, June 1964.
[10] Kohavi, Z.: "Reduction of output dependency in sequential machines," *IEEE Trans. Electron. Computers*, vol. EC-14, pp. 932–934, December 1965.
[11] Kohavi, Z., and E. J. Smith: "Decomposition of sequential machines," in *Proc. Sixth Ann. Symp. Switching Theory and Logical Design*, Ann Arbor, Mich., October 1965.
[12] Krohn, K. B., and J. L. Rhodes: "Algebraic theory of machines," in *Proc. Symp. Mathematical Theory of Automata*, Polytechnic Press, Brooklyn NY, 1962.
[13] Smith, E. J., and Z. Kohavi: "Synthesis of multiple sequential machines," in *Proc. Seventh Ann. Symp. Switching and Automata Theory*, Berkeley CA, October 1966.
[14] Stearns, R. E., and J. Hartmanis: "On the state assignment problem for sequential machines II," *IRE Trans. Electron. Computers*, vol. EC-10, no. 4, pp. 593–603, December 1961.
[15] Yoeli, M.: "The cascade decomposition of sequential machines," *IRE Trans. Electron. Computers*, vol. EC-10, pp. 587–592, April 1961.
[16] Yoeli, M.: "Cascade-parallel decompositions of sequential machines," *IEEE Trans. Electron. Computers*, vol. EC-12, no. 3, pp. 322–324, June 1963.
[17] Zeiger, H. P.: "Loop-free synthesis of finite-state machines," MIT. Ph.D. thesis, Dept of Electrical Engineering, Cambridge MA, September 1964.

Problems

Problem 12.1. Show that every n-state machine has N distinct state assignments, where

$$N = \frac{(2^k - 1)!}{(2^k - n)!k!}, \qquad k = \lceil \log_2 n \rceil.$$

Note that two assignments are said to be *distinct* if one cannot be obtained from the other by permuting or complementing the variables or by relabeling them.

Hint: Recall that k binary variables can be permuted in $k!$ ways and that there are 2^k ways of complementing them.

Problem 12.2

(a) Given the machine shown in Table P12.2 and two assignments α and β, derive in each case the logic equations for the state variables and output function and compare the results.

(b) Express explicitly in each case the dependency of the output and state variables.

Table P12.2

	NS		z	
PS	$x=0$	$x=1$	$x=0$	$x=1$
A	D	C	0	0
B	F	C	0	1
C	E	B	0	0
D	B	E	1	0
E	A	D	1	1
F	C	D	1	0

	$y_1y_2y_3$
A	000
B	001
C	010
D	011
E	100
F	101
Assignment α	

	$y_1y_2y_3$
A	110
B	101
C	100
D	000
E	001
F	010
Assignment β	

Problem 12.3. A six-state machine is said to have the five closed partitions shown below and *no* other closed partitions. Is this possible?

$$\pi_1 = \{\overline{A, C}; \overline{B}; \overline{D}; \overline{E, F}\}, \qquad \pi_4 = \pi(0),$$

$$\pi_2 = \{\overline{A, D}; \overline{B, C}; \overline{E}; \overline{F}\}, \qquad \pi_5 = \pi(I),$$

$$\pi_3 = \{\overline{A, B}; \overline{C, D}; \overline{E, F}\}.$$

Problem 12.4. The machine shown in Table P12.4 has the following closed partitions:

$$\pi_1 = \{\overline{A, C, E}; \overline{B, D, F}\}, \qquad \pi_2 = \{\overline{A, F}; \overline{B, E}; \overline{C, D}\}.$$

Table P12.4

	NS		
PS	$x=0$	$x=1$	z
A	D	C	1
B	A	D	0
C	B	E	0
D	E	B	0
E	F	C	0
F	C	D	0

(a) Find a state assignment that reduces the interdependencies of the state variables.
(b) Derive the logic equations and show the circuit diagram when unit delays are used as memory elements.

Problem 12.5

(a) Show that every closed partition is the sum of some *basic* partitions. (Recall that a *basic* partition $\pi_{S_i S_j}$ is the smallest closed partition containing $S_i S_j$ in one block.)
(b) Use the result of (a) to show that the procedure outlined in Section 12.3 for the construction of the π-lattice indeed gives all the closed partitions.

Problem 12.6. Let λ_o and λ'_o be two output-consistent partitions on the set of states of a machine M. Prove that $\lambda_o + \lambda'_o$ and $\lambda_o \cdot \lambda'_o$ are also output-consistent partitions.

Problem 12.7

(a) Let π be a closed partition on the set of states of a machine M. Prove that if π is also an output-consistent partition, i.e., $\pi \leq \lambda_o$, then M can be reduced to an equivalent machine that has only $\#(\pi)$ states. Conversely, if there are no closed partitions on M that are also output-consistent then M is in reduced form.
(b) Demonstrate the above reduction procedure by first finding a closed partition that is also output-consistent for the machine shown in Table P12.7 and *then* reducing it.

Table P12.7

	NS		
PS	$x=0$	$x=1$	z
A	E	C	0
B	B	A	1
C	B	D	0
D	E	C	1
E	E	F	1
F	B	C	0

Problem 12.8. The incompletely specified machine in Table P12.8 has a nontrivial closed partition that is also input-consistent. Does it have an autonomous clock? If yes, show its state diagram; if no, explain why not.

Table P12.8

	NS		
PS	I_1	I_2	I_3
A	—	A	—
B	C	—	D
C	A	B	A
D	B	A	B

Problem 12.9. In each of the following sets of partitions, π_1 and π_2 designate closed partitions while λ_o and λ_i designate output-consistent and input-consistent partitions, respectively.

(a) Construct the corresponding π-lattice for each case by obtaining all the necessary sums and products.

(b) Show schematic diagrams, demonstrating in each case the possible machine decompositions that yield minimal interdependencies of state variables as well as of outputs.

(i) $\pi_1 = \{\overline{A, B, E, F}; \overline{C, D, G, H}\}$, $\lambda_o = \{\overline{A, B, G, H}; \overline{C, D, E, F}\}$,
$\pi_2 = \{\overline{A, F, C, H}; \overline{B, D, E, G}\}$, $\lambda_i = \{\overline{A, C}; \overline{B, D}; \overline{E, G}; \overline{F, H}\}$,

(ii) $\pi_1 = \{\overline{A, B}; \overline{C, D}; \overline{E, F}; \overline{G, H}\}$, $\lambda_o = \lambda_i$,
$\pi_2 = \{\overline{A, E}; \overline{B, F}; \overline{C, G}; \overline{D, H}\}$, $\lambda_i = \{\overline{A, B, C, D}; \overline{E, F, G, H}\}$,

(iii) $\pi_1 = \{\overline{A, C, E, G}; \overline{B, D, F, H}\}$, $\lambda_0 = \{\overline{A, C}; \overline{B, D}; \overline{E, G}; \overline{F, H}\}$,
$\pi_2 = \{\overline{A, G}; \overline{B, F}; \overline{C, E}; \overline{D, H}\}$, $\lambda_i = 1$.

Problem 12.10

(a) For the machine shown in Table P12.10, find the π-lattice and obtain the input-consistent and output-consistent partitions.

Table P12.10

	NS		z	
PS	$x=0$	$x=1$	$x=0$	$x=1$
A	D	C	0	0
B	C	D	0	1
C	E	F	0	0
D	F	F	0	1
E	G	H	0	0
F	H	G	0	1
G	B	A	0	0
H	A	B	0	1

(b) Show two assignments that result in autonomous clocks of different frequencies. In each case, determine the period of the clock and draw a schematic diagram indicating the interdependencies within the decomposed machine.

Problem 12.11

(a) For the machine shown in Table P12.11, find λ_i and λ_o and construct the π-lattice.

(b) Choose as a basis for your state assignment three partitions, τ_1, τ_2, and τ_3 (which may or may not be closed), such that the following functional dependencies result:

$$Y_1 = f_1(y_1),$$
$$Y_2 = f_2(x, y_2, y_3),$$
$$Y_3 = f_3(x, y_2, y_3),$$
$$z = f_0(y_1, y_2).$$

Specify the desired relationship between the chosen τ's and λ_o and λ_i, and show a schematic diagram of the resulting structure.

(c) From the chosen τ's, obtain a state assignment and derive the corresponding logic equations.

Table P12.11

	NS		
PS	$x=0$	$x=1$	z
A	F	D	0
B	D	E	0
C	E	F	0
D	A	B	0
E	B	C	0
F	C	A	1

Problem 12.12

(a) Find a state assignment for the machine shown in Table P12.12 such that it will have the structure shown in Fig. P12.12.

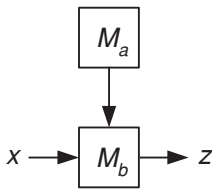


Fig. P12.12

Table P12.12

	NS		z	
PS	$x=0$	$x=1$	$x=0$	$x=1$
A	D	B	0	0
B	A	C	1	0
C	B	E	1	0
D	F	A	0	1
E	F	C	0	0
F	E	D	0	1

(b) Obtain the logic equations for the output function and state variables.

(c) Show the state diagram of the input-independent component.

Problem 12.13

(a) Find the π-lattice of the machine M shown in Table P12.13, and specify all the possible ways of decomposing the machine.

Table P12.13

	NS	
PS	$x=0$	$x=1$
A	B	C
B	C	D
C	D	C
D	E	B
E	D	A

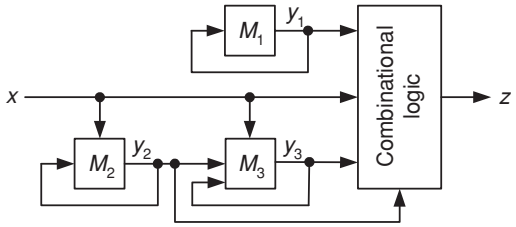
(b) Identify the states (A, B) and construct the implication graph. Augment the machine accordingly.
(c) Describe all the possible ways of decomposing the augmented machine M'. Specify in each case the dependencies of state variables.

Problem 12.14. The machine shown in Table P12.14 has the closed partition $\pi = \{\overline{A, C, D, F}; \overline{B, E, G}\}$.
(a) Can you find another closed partition such that a parallel decomposition is possible, without increasing the number of state variables?
(b) Construct an implication graph, starting with the vertex (A, B), and show that there exists a machine M', equivalent to M, that can be decomposed into the form shown in Fig. P12.14.

Table P12.14

	NS, z	
PS	$x = 0$	$x = 1$
A	$F, 1$	$C, 0$
B	$E, 0$	$B, 1$
C	$D, 0$	$C, 0$
D	$F, 1$	$C, 1$
E	$G, 0$	$B, 0$
F	$A, 1$	$F, 1$
G	$E, 1$	$G, 0$

Fig. P12.14



(c) Show the state tables of the component machines.
(d) Select an assignment that will lead to the structure of Fig. P12.14. Derive the corresponding logic equations.

Problem 12.15
(a) Prove that if τ is a partition on C_1 then

$$M\{m[M(\tau)]\} = M(\tau) \qquad \text{and} \qquad m\{M[m(\tau)]\} = m(\tau).$$

(b) Use the above to show that, for the partition τ of C_1,

$$\{M(\tau), m[M(\tau)]\} \qquad \text{and} \qquad \{M[m(\tau)], m(\tau)\}$$

are Mm pairs.

Problem 12.16. This problem is concerned with establishing a number of algebraic properties of Mm pairs and demonstrating that the set of all Mm pairs on a machine forms a lattice under the ordering defined in the text.

(a) Show that if $\lambda = M(\lambda')$ and $\tau = M(\tau')$ then $\lambda \cdot \tau = M(\lambda' \cdot \tau')$.
(b) Show that if $\lambda' = m(\lambda)$ and $\tau' = m(\tau)$ then $\lambda' + \tau' = m(\lambda + \tau)$.
(c) Prove that if (λ, λ') and (τ, τ') are Mm pairs then their glb and lub are given by

$$\text{glb}\{(\lambda, \lambda'), (\tau, \tau')\} = [\lambda \cdot \tau, m(\lambda \cdot \tau)]$$

and

$$\text{lub}\{(\lambda, \lambda'), (\tau, \tau')\} = [M(\lambda' + \tau'), \lambda' + \tau'].$$

Problem 12.17. Find the set of all Mm pairs for the machine M_8 (Table 12.15) and draw its Mm-lattice.

Problem 12.18

(a) Obtain the set of all Mm pairs for the machine shown in Table P12.18 and draw the corresponding Mm-lattice.
(b) Show a state assignment that results in the following functional dependencies:

$$Y_1 = f_1(x_1, x_2, y_1),$$
$$Y_2 = f_2(x_1, x_2, y_2, y_3),$$
$$Y_3 = f_3(x_1, x_2, y_1, y_2, y_3).$$

Table P12.18

	NS			
	x_1x_2			
PS	00	01	10	z
A	C	B	D	0
B	A	E	C	0
C	E	B	D	0
D	C	C	E	0
E	E	D	B	1

Problem 12.19

(a) Find all the m-partitions for the machine shown in Table P12.19.

Table P12.19

	NS				
	x_1x_2				
PS	00	01	11	10	z
A	A	A	D	A	1
B	C	C	D	A	0
C	D	A	A	A	0
D	B	A	D	B	0
E	E	C	A	B	0

(b) Select a number of m-partitions and find their corresponding M-partitions, such that they yield an assignment in which every variable depends on just one variable and the external input.
(c) Draw a schematic diagram of the resulting machine structure.

Problem 12.20. Construct an arbitrary machine with five or six states and three or four input symbols such that there exists at least one assignment that causes each state variable to be dependent only on the other variables and independent of itself, that is, Y_1 is independent of y_1, etc.

Problem 12.21. The machine shown in Table P12.21 can be serially decomposed into three components without any increase in the number of state variables.
(a) Determine the period of the maximal autonomous clock.
(b) Select a set of partitions which induces an assignment such that the above serial decomposition is accomplished and the output logic is minimized.
(c) Show the state table of each component.

Table P12.21

	NS		z	
PS	$x=0$	$x=1$	$x=0$	$x=1$
A	D	C	0	0
B	C	D	0	1
C	E	F	0	0
D	F	F	0	1
E	G	H	0	0
F	H	G	0	1
G	B	A	0	0
H	A	B	0	1

Problem 12.22. The machine shown in Table P12.22 has the following partitions:

$$\pi_1 = \{\overline{A, B, C}; \overline{D, E, F}\}, \qquad \lambda_o = \{\overline{A, D, E}; \overline{B, C, F}\},$$
$$\pi_2 = \{\overline{A, F}; \overline{B, E}; \overline{C, D}\}, \qquad \lambda_i = \{\overline{A, C}; \overline{B}; \overline{D, F}; \overline{E}\}.$$

Table P12.22

	NS		NS	
PS	$x=0$	$x=1$	$x=0$	$x=1$
A	E	E	0	0
B	D	F	0	1
C	F	D	0	1
D	A	C	0	0
E	C	A	0	0
F	B	B	0	1

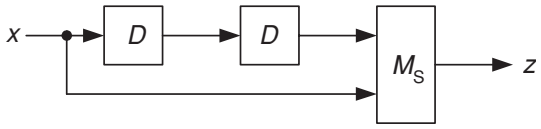
(a) Draw a schematic diagram of the machine's structure induced by these partitions.
(b) Show complete state tables for the component machines.

Problem 12.23. The machine of Table P12.23 is to be realized in the form shown in Fig. P12.23, where each block designated D represents a pure delay without internal feedback. Find a state table for a successor machine M_S such that the number of state variables and the functional complexity of the output are minimized.

Table P12.23

	NS		NS	
PS	$x=0$	$x=1$	$x=0$	$x=1$
A	E	A	0	0
B	D	B	0	1
C	D	B	0	0
D	F	C	1	1
E	E	C	1	0
F	F	B	1	1

Fig. P12.23



Problem 12.24. Prove that if two machines M_1 and M_2 are reduced then, for specified initial states, the composite machine is also reduced.

Problem 12.25. The machine M_1 shown in Table P12.25 is to be realized in a cascade form, with a machine M_2 as the predecessor component. The starting states are A and P.
(a) Show the state table of an appropriate successor component.
(b) Choose a state assignment for M_1 that preserves the above structure and, at the same time, minimizes the complexity of the output function.
(c) Derive the logic equations for the state variables and output function.

Table P12.25

	NS		z	
PS	$x=0$	$x=1$	$x=0$	$x=1$
A	B	E	0	1
B	D	C	1	1
C	G	C	0	0
D	E	F	0	0
E	B	A	0	1
F	C	D	1	1
G	F	E	0	0

M_1

	NS	
PS	$x=0$	$x=1$
P	R	Q
Q	R	P
R	S	Q
S	Q	S

M_2

Problem 12.26. The machine M of Table P12.26 is to be realized in the form of Fig. P12.26. The state transitions of the component M_a are specified as shown. The starting state of M is A and that of M_a is G. Find the state table of M_b and specify the combinational logic that generates z.

Table P12.26

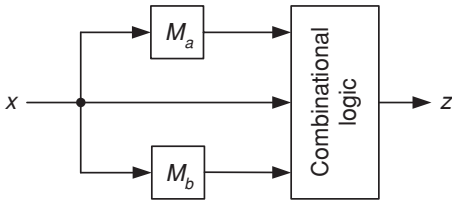
	NS, z	
PS	$x=0$	$x=1$
A	$B, 0$	$C, 0$
B	$C, 0$	$D, 1$
C	$D, 1$	$E, 1$
D	$E, 0$	$F, 1$
E	$F, 1$	$A, 0$
F	$A, 1$	$B, 1$

M

	NS	
PS	$x=0$	$x=1$
G	H	G
H	G	H

M_a

Fig. P12.26



Problem 12.27. The machines M_1 and M_2 of Table P12.27 can be jointly realized in the form shown in Fig. P12.27, with only three state variables.

Table P12.27

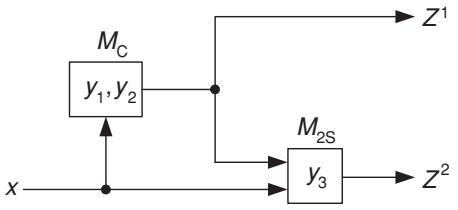
	NS		
PS	$x=0$	$x=1$	Z^1
P	Q	R	0
Q	P	Q	0
R	Q	P	1

M_1

	NS		
PS	$x=0$	$x=1$	Z^2
A	B	D	0
B	E	C	0
C	A	B	0
D	B	A	1
E	C	E	1

M_2

Fig. P12.27



(a) Construct a composite machine from M_1 and M_2 when the initial states are P and A for M_1 and M_2, respectively.
(b) Show the state tables for M_C and M_{2S}. Use the state names $S_1, S_2, \ldots$ and $R_1, R_2, \ldots$, etc.
(c) Show the logic equations for the outputs.

Problem 12.28. Consider the machines M_1 and M_2 shown in Table P12.28. Their starting states are R_1 and S_1, respectively.
(a) Find the π-lattice for each machine and determine whether a common predecessor machine exists.
(b) Show that if the state S_2 is split into S_2' and S_2'', a common predecessor can be found.
(c) Realize the two machines in the form shown in Fig. 12.18. Show the state tables of the predecessor and successor machines.

Table P12.28

	NS		Z^1	
PS	$x=0$	$x=1$	$x=0$	$x=1$
R_1	R_2	R_4	1	0
R_2	R_1	R_3	0	1
R_3	R_1	R_4	0	1
R_4	R_2	R_3	1	0

M_1

	NS		Z^2	
PS	$x=0$	$x=1$	$x=0$	$x=1$
S_1	S_1	S_3	0	0
S_2	S_1	S_2	0	1
S_3	S_2	S_3	1	1

M_2

Problem 12.29. The disjoint realization of machines M_1 and M_2 shown in Table P12.29, requires six state variables. Find another realization for these machines that requires just four state variables and has the form shown in Fig. P12.29. Assume that

Table P12.29

	NS		
PS	$x=0$	$x=1$	Z^1
S_1	S_6	S_3	0
S_2	S_5	S_2	0
S_3	S_4	S_3	0
S_4	S_6	S_2	0
S_5	S_7	S_2	0
S_6	S_1	S_6	0
S_7	S_5	S_7	1

M_1

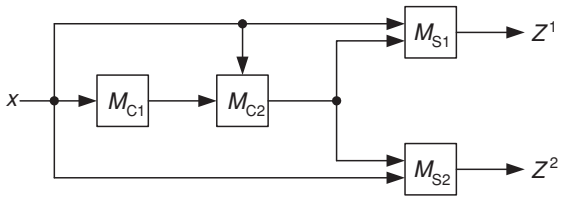
	NS		
PS	$x=0$	$x=1$	Z^2
Q_1	Q_3	Q_4	0
Q_2	Q_4	Q_5	0
Q_3	Q_1	Q_3	0
Q_4	Q_2	Q_4	0
Q_5	Q_6	Q_5	0
Q_6	Q_3	Q_4	1

M_2

states S_1 and Q_1 are the initial states. Show the state table of each component and indicate the functional dependencies of the outputs.

Hint: You may find it necessary to split some states.

Fig. P12.29



Problem 12.30. Repeat Problem 12.29 for the machine M_1 shown in Table P12.30 and the machine M_2 shown in Table P12.29.

Hint: It is quite straightforward to find a common-factor machine that has two states. However, if you construct the composite machine for M_1 and M_2 and draw its implication graphs for the initial identifications (S_1Q_1, S_2Q_1) and (S_1Q_1, S_1Q_2), you can show that a common-factor machine that has four states can be found, while each of the successors has only two states.

Table P12.30

	NS		
PS	$x=0$	$x=1$	Z^1
S_1	S_5	S_4	1
S_2	S_5	S_3	0
S_3	S_1	S_3	0
S_4	S_2	S_4	0
S_5	S_2	S_5	1

CHAPTER

13 State-identification experiments and testing of sequential circuits

In this chapter, we shall be concerned with experimental analysis of the behavior of finite-state machines, test generation for sequential circuits, design for testability, and built-in self-test (BIST).

A machine will be assumed to be reduced, strongly connected, and completely specified. State-identification experiments are designed to identify the unknown initial state of the machine and, whenever such an identification is unnecessary or impossible, to identify the final state of the machine. These experiments are known as distinguishing and homing experiments, respectively. Machine-identification experiments are concerned with the problem of determining whether a given n-state machine is distinguishable from all other n-state machines. This problem is shown to be, under certain conditions, equivalent to the problem of determining whether a given machine is operating correctly.

Test generation methodologies will be presented for sequential circuits under two fault models: functional and stuck-at. A *functional fault* alters the machine's state table. A *stuck-at fault* is manifested as a permanent 0, i.e., a stuck-at-0 (s-a-0) fault, or as a permanent 1, i.e., a stuck-at-1 (s-a-1) fault on some line in the circuit, as discussed in Chapter 8. Since there is no direct way to control the present state lines of a sequential circuit or observe its next state lines, sequential test generation is a difficult task. To ease the testing burden, one can use design-for-testability methods, such as scan design, to allow the control and observation of state lines. Another way to reduce the testing burden is to allow the circuit to test itself through the BIST method.

13.1 Experiments

The application of an input sequence to the input terminals of a machine is referred to as an *experiment* on the machine. An experiment designed to take the machine through all its transitions, in such a way that a definite conclusion can be reached as to whether the machine is operating correctly, is said to be a *checking experiment*. At the beginning of an experiment, the machine is said to

be in an *initial* (or *starting*) state and at the end of an experiment the machine is said to be in a *final* state. It is customary to distinguish between two types of experiments:

1. *simple experiments*, which are performed on a single copy of the machine;
2. *multiple experiments*, which are performed on two or more identical copies of the machine.

In practice, most machines are available in just a single copy, and therefore simple experiments are preferable to multiple ones.

Experiments are classified according to their performance as:

1. *adaptive experiments*, in which the input symbol at any instant of time depends on the previous output symbols;
2. *preset experiments*, in which the entire input sequence is predetermined independently of the outcome of the experiment.

Since preset experiments are simpler to perform in today's technology, we shall focus on such experiments.

A measure of the efficiency and cost of an experiment is its *length*, which is the total number of input symbols applied to the machine during the execution of the experiment.

In Chapter 10 we studied the properties of experiments used to distinguish between two nonequivalent states, S_i and S_j, of an n-state machine. We showed that if S_i and S_j are distinguishable then they can be distinguished by an experiment of length at most $n - 1$. We now consider more general problems, that of identifying the initial or final state of a given machine and that of distinguishing a given n-state machine from all other n-state machines that have the same input and output alphabets.

Introductory example

Consider the machine M_1 (Table 13.1), which may initially be in any of the states A, B, C, or D. The responses of M_1 to the input sequences 01 and 111 are listed in Table 13.2. Knowing the output sequence that M_1 produces in response to input sequence 01 is always sufficient to determine uniquely M_1's *final* state, since each of the output sequences that might result from the application of 01 is associated with just one final state. For example, output sequence 00 indicates that the final state is B, while output sequences 11 or 01 indicate that the final state is D or A, respectively. On the other hand, the knowledge of the response of M_1 to input sequence 01 is not sufficient to determine M_1's *initial* state, since the production of output sequence 00 could mean that the initial state was A or that it was B. In fact, if M_1 was initially in either state A or B, it is impossible to determine the initial state by an experiment which starts with a 0, since the 0-successors of both A and B are C, and the output symbol

Table 13.1 Machine M_1

	NS, z	
PS	$x = 0$	$x = 1$
A	$C, 0$	$D, 1$
B	$C, 0$	$A, 1$
C	$A, 1$	$B, 0$
D	$B, 0$	$C, 1$

Table 13.2 Responses of M_1 to the input sequences 01 and 111

Initial state	Response to 01	Final state
A	00	B
B	00	B
C	11	D
D	01	A

(*a*)

Initial state	Response to 111	Final state
A	110	B
B	111	C
C	011	D
D	101	A

(*b*)

produced in both cases is 0. No sequence following the initial 0 input symbol will yield any new information regarding the initial state.

Using the same line of argument, it is evident that the output sequence that M_1 produces in response to input sequence 111 is always sufficient to determine uniquely M_1's final state, as well as its initial state. As shown in Table 13.2, each of the output sequences that might result from the application of 111 to M_1 is associated with just one initial state and one final state.

Before presenting techniques to be used in the design of experiments, we shall introduce some terminology and define the successor tree, which will prove to be an effective tool in the design of minimal experiments.

Uncertainties

Suppose that a machine M, which is given to the experimenter, can initially be in any of its n states. In such a case, we say that the initial uncertainty regarding the state of the machine is given by $(S_1 S_2 \cdots S_n)$. Thus, the *initial uncertainty* is the minimal subset of S (including S itself) that is known to contain the initial state. For example, if the machine M_1 can initially be in any of its four states then the initial uncertainty is $(ABCD)$.

Our aim is to perform experiments that reduce the initial uncertainty and, whenever possible, reveal the initial or final state. For example, suppose that we apply an input symbol 1 to machine M_1 and that in response it produces the output symbol 0. We may conclude that M_1 was initially in state C, since only from that state is a response of 0 to input symbol 1 possible. The final state in this case is B. However, suppose the response of M_1 to input symbol 1 is 1;

then all we can say regarding the final state of the machine is that it may be any of the states D, A, or C, depending on whether the initial state was A, B, or D, respectively. The set of states (ACD) thus represents the uncertainty regarding the final state of M_1 after the application of the input symbol 1. In general, the *uncertainty* regarding the state of M after the application of X is a specific subset of the X-successors of the states contained in the initial uncertainty. The elements of the uncertainty are not necessarily distinct.

Let U_0 be the initial uncertainty, and let input symbol I_i result in an uncertainty U_i; then U_i is said to be the I_i-*successor* of U_0. Suppose, for example, that the initial uncertainty regarding the state of M_1 is (ACD). If an input symbol 1 is now applied to M_1, the successor uncertainty will be (B) or (CD), depending on whether the output symbol is 0 or 1, respectively. We thus say that the uncertainties (B) and (CD) are the 1-successors of (ACD). Subsequently, we shall refer to a collection of uncertainties as an *uncertainty vector*. The individual uncertainties contained in the vector are called the *components* of the vector. An uncertainty vector whose components contain a single state each is said to be a *trivial uncertainty vector*. An uncertainty vector whose components contain either single states or identical repeated states is said to be a *homogeneous uncertainty vector*. Thus, for example, the vectors $(AA)(B)(C)$ and $(A)(B)(A)(C)$ are homogeneous and trivial, respectively.

The successor tree

The *successor tree*, which is defined for a specified machine M and a given initial uncertainty, displays graphically the I_i-successor uncertainties for all I_i and thus assists the experimenter in the selection of the most suitable input sequence. It is composed of branches arranged in successive *levels*, numbered $0, 1, \ldots, j, \ldots$ Each branch in the jth level splits into p branches, labeled I_1, $I_2, \ldots, I_p$, corresponding to the input symbols of the machine. The branches emanating from the jth level form the $(j+1)$th level, and so on. Each node of the successor tree is associated with an uncertainty vector. The highest node (in level 0) is associated with initial uncertainty U_0, and each of the p nodes in level 1 is associated with a successor of U_0. The jth level of the tree consists of p^j branches, each terminating at a node. A sequence of j branches, starting at the highest node and terminating at a node in the jth level, is referred to as a *path* in the tree; j is called the *length* of the path. Each path *describes* an input sequence which, when applied to the machine, results in the uncertainty vector associated with the terminal node in the jth level. Hence, a tree with $j+1$ levels contains p^j paths, describing the p^j input sequences of length j.

The successor tree for the machine M_1 and an initial uncertainty $(ABCD)$ is shown in Fig. 13.1. It contains four levels numbered 0 through 3. Each branch is labeled with the input symbol that it represents, and every node is associated with the corresponding uncertainty vector. The highest node is associated with

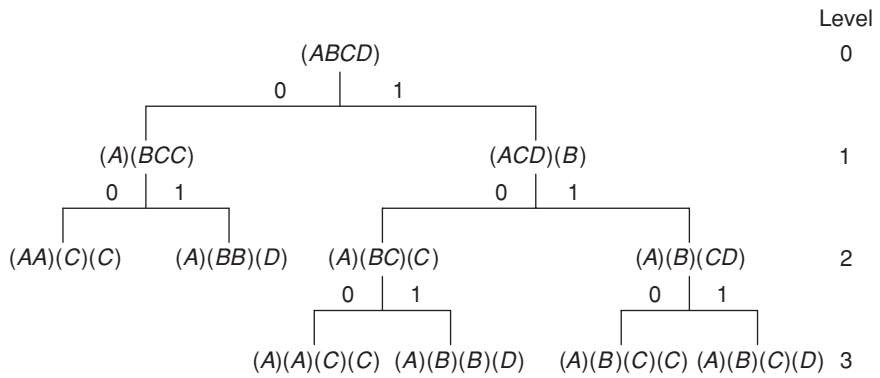


Fig. 13.1 Successor tree for M_1.

the initial uncertainty while the nodes in level 1 are associated with its 1- and 0-successors, and so on. For example, an input symbol 1 applied to M_1 when the initial uncertainty is $(ABCD)$ results in the uncertainty vector $(ACD)(B)$, while an input symbol 0 results in the uncertainty vector $(A)(BCC)$. The 1-successor of the vector $(ACD)(B)$ is determined by obtaining the 1-successors of (ACD) and (B) separately. For example, the 1-successor of (B) is (A), since the application of an input symbol 1 to M_1, when in state B, takes it to state A. The 1-successor of (ACD), however, depends on the output symbol; it is (CD) if the output symbol is 1, and (B) if it is 0. Thus, the corresponding uncertainty vector is $(A)(B)(CD)$. Similarly, the 0-successor of $(ACD)(B)$ is $(A)(BC)(C)$, since the 0-successor of (B) is (C) while that of (ACD) is $(A)(BC)$.

An uncertainty is said to be *smaller* than another uncertainty if it contains fewer elements; e.g., (BC) is smaller than (ACD). From the way in which the tree is constructed, it is evident that an uncertainty associated with a node in the jth level is either smaller than or contains the same number of elements as its predecessor in the $(j-1)$th level. A homogeneous uncertainty vector will always have as its successors homogeneous uncertainty vectors. For example, in the tree of machine M_1 the successors of the uncertainty (BCC) are $(AA)(C)$ and $(A)(BB)$. The tree may be continued as far as is necessary but, for it to be of practical value, a truncated version must be defined by stipulating a number of termination rules.

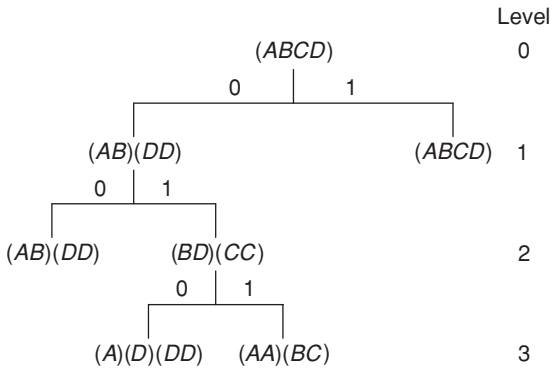
13.2 Homing experiments

The objective of this section is to develop techniques for the construction of experiments to identify the final state of a given n-state machine. It is shown that such experiments can be constructed for every reduced machine, and bounds on their lengths are derived.

Table 13.3 Machine M_2

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 0$	$D, 0$
B	$A, 0$	$B, 0$
C	$D, 1$	$A, 0$
D	$D, 1$	$C, 0$

Fig. 13.2 Homing tree for M_2.



Definition 13.1 An input sequence Y_0 is said to be a *homing sequence* if the final state of the machine can be determined uniquely from the machine's response to Y_0, regardless of the initial state.

The homing tree

A homing sequence for a given machine M may be obtained from a truncated version of its successor tree. Our task is to construct the tree and obtain the shortest path leading from the initial uncertainty to a trivial uncertainty or a homogeneous uncertainty. The presence of such an uncertainty at the kth level of the tree guarantees that there exists an input sequence consisting of k symbols whose application to M is sufficient to specify uniquely M's final state.

A *homing tree* is a successor tree in which a jth-level node becomes terminal when either of the following occur:

1. the node is associated with an uncertainty vector whose nonhomogeneous components are associated with some node in a preceding level;
2. some node in the jth level is associated with a trivial or homogeneous vector.

The homing tree of a machine M_2 (Table 13.3) is shown in Fig. 13.2. The node associated with the vector $(AB)(DD)$ in level 2 is a terminal node, since its predecessor in level 1 is also associated with vector $(AB)(DD)$.

Table 13.4 The response of M_2 to the homing sequence 010

Initial state	Response to 010	Final state
A	000	A
B	001	D
C	101	D
D	101	D

Similarly, the node $(ABCD)$ in level 1 is terminated, since it is identical with the node $(ABCD)$ in level 0. The nodes in level 3 are also terminal nodes, since $(A)(D)(DD)$ is a homogeneous uncertainty vector. The shortest homing sequence is 010, since it is the shortest sequence described by a path leading from the zeroth level to a homogeneous uncertainty. The response and final states corresponding to this sequence are given in Table 13.4.

We shall now establish the existence of the homing experiment and derive a bound on its length.

Theorem 13.1 *A preset homing sequence, whose length is at most* $(n-1)^2$, *exists for every reduced n-state machine M.*

Proof Let the initial uncertainty be $(S_1 S_2 \cdots S_n)$. Since M is reduced, for every pair of states S_i, S_j there exists an experiment (i.e., a sequence) of length $n-1$ or shorter that distinguishes S_i from S_j. Let us denote this experiment as λ_k. Starting at the initial uncertainty, application of sequence λ_1, which distinguishes between some pair of states in M, yields the λ_1-successor uncertainty vector, which contains at least two components. Next, we select any two states in one component and apply the appropriate sequence λ_2, which distinguishes between them. The $\lambda_1\lambda_2$-successor uncertainty vector contains at least three components. In a similar manner, we obtain the $\lambda_1\lambda_2\cdots\lambda_{n-1}$-successor vector, which consists of n components, each containing only one state. Therefore, the sequence $\lambda_1\lambda_2\cdots\lambda_{n-1}$ is a homing sequence whose length is at most $(n-1)^2$. ◇

This value is an upper bound on the length of the homing sequence, but is not the least upper bound. It can be shown that the length of the homing sequence need not exceed $\frac{1}{2}n(n-1)$ and that this is indeed a tight bound (see Problem 13.5).

Synchronizing experiments

A *synchronizing sequence* of a machine M is a sequence that takes M to a specified final state, regardless of the output symbols or initial state. Some machines possess such sequences; others do not.

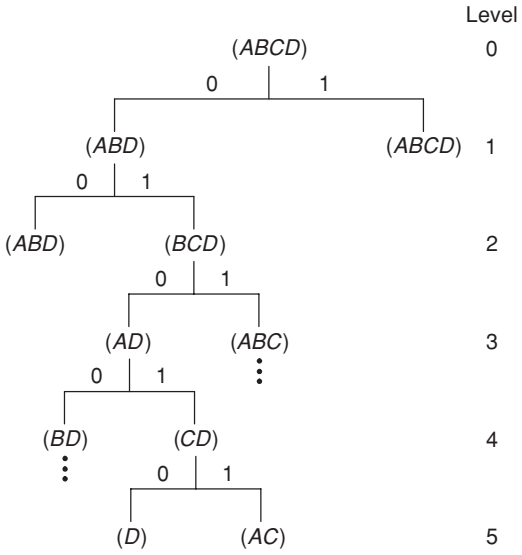


Fig. 13.3 Synchronizing tree for M_2.

For a given machine, we can construct a successor tree by ignoring the output symbols and associating with every node in the jth level the uncertainty regarding the final state resulting from the application of the first j input symbols. For example, if the initial uncertainty of the machine M_2 is $(ABCD)$ then the 0-successor uncertainty is (ABD), and so on. Note that, since we are interested only in the final state regardless of the output symbols, it is not necessary to write down repeated entries; e.g., $(ABDD)$ may be simply written as (ABD), etc. A jth-level node in the tree becomes terminal whenever either of the following occurs:

1. the node is associated with an uncertainty that is also associated with some node in a preceding level;
2. some node in the jth level is associated with an uncertainty containing just a single element.

A tree so constructed will be called a *synchronizing tree*. The synchronizing tree for the machine M_2 is shown in Fig. 13.3.

A synchronizing sequence is described by (corresponds to) a path in the tree leading from the initial uncertainty to a singleton uncertainty, i.e., an uncertainty containing just a single state. For the machine M_2, the path 01010 describes a synchronizing sequence that, when applied to M_2, synchronizes the machine to state D regardless of the output symbols or initial state. Note that if the initial uncertainty of M_2 is (BCD) then the sequence 010 synchronizes M_2 to state D, since the 010-successors of B, C, and D are D, as shown in Table 13.4.

Theorem 13.2 *If a synchronizing sequence for an n-state machine M exists then its length is at most* $\frac{1}{2}(n-1)^2 n$.

Proof Let the initial uncertainty be $(S_1 S_2 \cdots S_n)$. Select any two states S_i, S_j and apply to them a sequence ξ_1 that takes them into some state S_k. This task can always be accomplished, since M is known to possess a synchronizing sequence. The length of the sequence ξ_1 is at most $\frac{1}{2}(n-1)n$, since the longest path for the synchronization of $(S_i S_j)$ is through all possible pairs of states, i.e., $(S_1 S_2)$, $(S_1 S_3)$, $\ldots$, $(S_{n-1} S_n)$. Consequently, S_k is the ξ_1-successor of $(S_i S_j)$. Next, select a state S_p from the resultant uncertainty, and determine the sequence ξ_2 that takes $(S_k S_p)$ into some state S_q. The length of ξ_2 is also at most $\frac{1}{2}(n-1)n$. In the same way, it is possible to find sequences $\xi_3, \xi_4, \ldots, \xi_{n-1}$, which, when concatenated, yield the synchronizing sequence $\xi_1 \xi_2 \cdots \xi_{n-1}$, whose length is at most $\frac{1}{2}(n-1)^2 n$. ◇

The above bound is not the least upper bound. For a tighter bound, see Appendix 13.1.

13.3 Distinguishing experiments

Distinguishing experiments are concerned with the identification of the initial state of a machine whose state table is known but about which there is no other information regarding its condition.

Definition 13.2 Let M be an n-state machine. An input sequence X_0 is said to be a *distinguishing sequence* if the output sequence produced by M in response to X_0 is different for each initial state.

Knowing the output sequence that M produces in response to X_0 is sufficient to identify uniquely M's initial state. However, knowledge of the initial state and the input sequence is always sufficient to determine uniquely the final state as well. Consequently, *every distinguishing sequence is also a homing sequence*. The converse, however, is not true, since many homing sequences do not provide all the information regarding the initial state, e.g., the sequence 010 for machine M_2.

The distinguishing tree

A *distinguishing tree* is a successor tree in which a node in the jth level becomes terminal when any of the following occurs:

1. the node is associated with an uncertainty vector whose nonhomogeneous components are associated with some node in a preceding level;
2. the node is associated with an uncertainty vector containing a homogeneous nontrivial component;
3. some node in the jth level is associated with a trivial uncertainty vector.

A path in the tree describes a distinguishing sequence of M if and only if it starts in the initial uncertainty (which is assumed to consist of the entire set of

states S) and terminates in a node associated with a trivial uncertainty. A bound on the length of distinguishing sequences is shown in Appendix 13.2.

The distinguishing tree of the machine M_1 is obtained from the corresponding successor tree (Fig. 13.1). The node associated with the homogeneous uncertainty vector $(A)(BCC)$ is terminated, since no further experiment can split the component (CC); i.e., there is no way of knowing, once the machine has passed to state C, whether the initial state was A or B. The machine M_1 has four distinguishing sequences of length 3, 111, 110, 101, and 100. The response of M_1 to the sequence 111 is summarized in Table 13.2*b*. This sequence clearly causes four distinct responses, depending on the initial state.

While every machine has at least one homing sequence, not every machine has a distinguishing sequence. For example, the distinguishing tree of the machine M_2 must be terminated in level 1 (see Fig. 13.2), since the vector $(ABCD)$ is identical to the initial uncertainty and the vector $(AB)(DD)$ has a nontrivial homogeneous component. An inspection of the state table of M_2 (Table 13.3) would have revealed the same result, since no experiment that starts with an input symbol 0 will distinguish between states C and D or between states A and B, while no experiment that starts with a 1 will reduce the initial uncertainty.

The shortest distinguishing prefix

In many cases, the initial state of a machine can be determined just from the prefix of distinguishing sequence X_0. The length of the required prefix is a function of the initial state. Consider again the machine M_1, whose response to the distinguishing sequence 111 is given in Table 13.2*b*. It is evident that if the response of the machine to the first input symbol is 0 then the initial state must have been C, and the distinguishing experiment may be terminated at this stage. However, if the response is 1 then the initial state could have been either A, B, or D. The experiment must continue, and M_1 is supplied with a second input symbol 1. If M_1's response is now 0 then the initial state must have been D, and the distinguishing experiment may be terminated. If, however, the response is 1 then the uncertainty regarding the initial state is (AB) and a third input symbol 1 must be applied to the machine. Thus, for the machine M_1 and the distinguishing sequence 111, the shortest distinguishing prefix for state C is 1, for state D 11, and for states A and B 111.

The shortest distinguishing prefixes can be determined by means of a modified distinguishing tree (see [9]). They are particularly useful in checking experiments and machine identification, where they lead to relatively short experiments.

13.4 Machine identification

Up to now we have been concerned with the problems of identifying the initial and final states of a known machine. We shall now address ourselves to a

more general problem – that of identifying an unknown machine. The machine identification problem is essentially that of experimentally determining the state table of an unknown machine. In its most general form, when no information is available on the unknown machine, this problem cannot be solved for several reasons. First, the experimenter must have complete information regarding the input alphabet of the machine, since otherwise he or she can never be sure that the next input symbol will not reveal new information regarding the machine. Similarly, the machine cannot be identified unless there is an upper bound on the number of its states since, for any given machine and any experiment of length L, it is possible to construct another machine that responds to the experiments of length L exactly like the given machine but will respond differently to experiments of length greater than L. Finally, if a given machine M_i is in initial state S_i then it is indistinguishable experimentally from a machine M_j whose initial state S_j is equivalent to S_i, although machines M_i and M_j may, in fact, be distinguishable. This situation clearly will not occur if both M_i and M_j are strongly connected.

To make the problem of machine identification solvable, we impose several restrictions on the machines. We assume that the input alphabet is known, as is an upper bound on the number of states of the machine. Moreover, the machine is assumed to be reduced and strongly connected.

An unknown machine with at most n states can now be identified in the following manner. Construct the direct-sum table (see Problem 10.10) from all tables that have n or fewer states and find a homing sequence for it. Clearly such a homing sequence can always be found, and its application to the machine in question will reveal which set of equivalent states from the direct-sum table contains the final state of the machine. Also, if the direct-sum table contains only those tables that correspond to reduced and strongly connected machines, the homing sequence will uniquely identify the final state of the machine and, in turn, the machine itself. This demonstrates that, under specified conditions, in principle the machine identification problem can be solved. However, as a procedure for actually designing experiments the direct-sum approach is impractical, since the number of distinct tables is staggeringly large even for relatively small n's. It will be shown subsequently that the problem of devising checking experiments for sequential machines is directly related to the machine identification problem. More efficient procedures will be presented for the design of such experiments directly from the state table, without the use of the direct sum.

As an example, suppose that a machine is known to have two states and that its response to input sequence X is output sequence Z, as shown below.

Time :	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
Input, X :	1	1	1	0	1	0	1	
Output, Z :	0	1	0	0	1	0	0	

The first step in the analysis of these sequences is the identification of the distinct states of the tested machine. Let us name these two states A and B and

Table 13.5 Machine M_3

	NS, z	
PS	$x = 0$	$x = 1$
A	$A, 0$	$B, 0$
B	$B, 0$	$A, 1$

suppose that, at the start of the experiment, the machine was in state A. The application of an input symbol 1 results in an output symbol 0 and a transition that is yet to be determined. However, since the second input symbol is also a 1 but the response is 1, the machine must have been in a state other than A at t_2. Hence, the experimenter may conclude that at t_2 the machine was in state B.

Since state A is the only state which responds to an input symbol 1 by producing an output symbol 0, it is evident that at t_3 the machine was in state A. At t_4, it was again in state B, since it has already been verified (at t_2) that an input symbol 1 causes a transition from state A to B. In a similar manner, it is easy to show that at t_5 the machine was again in state B, which, in turn, implies (see t_3) that at t_6, it was in state A. Finally, at t_7, it must have been in state A, since this is the only state in which the machine produces a 0 output symbol as a response to a 1 input symbol. As a result of the above analysis, the experimenter is able to demonstrate that the machine indeed has two states, named A and B, and that its transitions and output symbols are given by the state table of Table 13.5. Thus, the above experiment is an identification experiment for a machine M_3.

13.5 Checking experiments

The problem of designing *checking experiments* is actually a restricted version of the problem of machine identification. An experimenter is supplied with a machine and its state table. The task is to determine from terminal experiments whether the given table accurately describes the behavior of the machine; that is, to decide whether the actual machine is isomorphic to the one described by the state table. As discussed before, we shall restrict our attention to machines that are strongly connected, completely specified, and reduced. We also assume that any faults are permanent, owing to some defect. This assumption excludes transient errors due to noise or incorrect input symbols.

First, we consider machines that possess at least one distinguishing sequence. In subsequent sections, we shall relax this restriction and discuss machines that have no distinguishing sequence. Note that these experiments are intended to detect the presence of one or more faults but will not locate or diagnose them.

Table 13.6 Machine M_4

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 0$	$C, 1$
B	$C, 0$	$D, 0$
C	$D, 1$	$C, 1$
D	$A, 1$	$B, 0$

Table 13.7 Responses of M_4

Initial state	Response to 00	Final state
A	00	C
B	01	D
C	11	A
D	10	B

(*a*)

Initial state	Response to 01	Final state
A	00	D
B	01	C
C	10	B
D	11	C

(*b*)

We will make the assumption that the machine either has a synchronizing sequence or a reset input that can transfer it to the initial state.

Designing checking experiments

In the procedure we use, each checking experiment consists of two parts.

1. The first part uses the synchronizing sequence or a reset input to transfer the machine into a prespecified state, which is the initial state for the second part of the experiment.
2. The second part is a *preset* experiment in which the machine is taken through all possible transitions. This part is subdivided into two subparts. In the first subpart the machine is caused to display the response of each of its states to the distinguishing sequence, while in the second subpart the actual transitions are verified.

As an example, consider the machine M_4 whose state table is given in Table 13.6 and whose responses to the sequences 00 and 01 are summarized in Table 13.7. Suppose that the synchronizing sequence or reset input places the machine in state A, from which the preset part of the experiment can commence.

In designing the preset part of the checking experiment, the first task is to ascertain that the starting state is indeed A and that the machine being tested actually contains four distinct states. This can be accomplished by displaying the response of each state to the same distinguishing sequence. The machine M_4 has two distinguishing sequences, 00 and 01, whose applications to the machine result in the responses shown in Table 13.7. The design of experiments based

on distinguishing sequence 00 is somewhat shorter but will be left to the reader as an exercise.

To display the response of the starting state, we apply the distinguishing sequence $X_0 = 01$. If the machine has operated correctly up to this point, its output response is 00 and it is now in state D. To display the response of this state, the distinguishing sequence X_0 is applied again and, as a result, the machine goes to state C. The application of a third distinguishing sequence leaves the machine in state B and displays the response of state C. Applying X_0 twice more leaves the machine in state B, as shown below:

Input :		0	1		0	1		0	1		0	1		0	1	
State :	*A*			*D*			*C*			*B*			*C*			*B*
Output :		0	0		1	1		1	0		0	1		1	0	

The first eight symbols, by displaying four different responses to input sequence 01, i.e., 00, 11, 10, and 01, verify that the machine in question indeed has four distinct states. The last two symbols guarantee that the machine terminates in state B, since it has already been established that a response of 10 to the distinguishing sequence indicates a transition from state C to state B. The above sequence thus verifies the existence of at least four states and, since we are assuming that M_4 has no more than four states, each state must have been visited at least once, and its response to the distinguishing sequence determined. From this point on, if at any time during the course of the experiment one of the above responses to the distinguishing sequence is produced, the state of the machine at that time is uniquely identifiable. (It must be emphasized that the names given to the states are of no importance; a different set of names would result in an isomorphic machine.)

If the machine has not produced the expected output sequence up to this point, we may conclude that a fault exists. If, however, the above expected output sequence has been produced then no conclusion can be reached as to whether the machine has operated correctly and is indeed in state B or a fault exists and the actual final state is different from B. We, therefore, assume for the present that the machine actually started in state A and terminated in B. If this assumption is incorrect, it will be revealed as such in the next part of the experiment.

To complete the experiment it is now necessary to verify every state transition. The general procedure to be followed is to apply the input symbol that causes the desired transition and to identify it by applying the distinguishing sequence. Since the machine is in state B, we shall start by applying an input symbol 0, followed by a distinguishing sequence 01. This input sequence takes the machine back to state B, and thus a 101 input sequence is applied to check the transition from B to D under a 1 input symbol and verify that the machine actually has moved to state D. In each of these three-bit sequences, the first bit causes the transition, while the distinguishing sequence ascertains that the transition is indeed the assumed one. At this point we have obtained additional

information about another transition. It has earlier been shown that the application of 01 to the machine while in state B causes it to go to state C. However, since input symbol 0 itself takes the machine from B to C, we may conclude that if a 1 input symbol is applied to the machine while in state C then it stays in state C. In other words, since the 01-successor of state B is C and the 0-successor of B is also C, the 1-successor of C must be C.

At this point, the machine is in state C. If, in response to the input sequence 001, the machine produces an output sequence 111, we may conclude that the 0-successor of C is D and that the final state is again C. However, since it has already been established that the 01-successor of C is B, it means that the 1-successor of D is B. The experiment at this stage is as follows; note that the second and third rows continue the first row.

Input :		0		1		0		1		0		1		0		1		0		
State :	*A*				*D*				*C*		*D*		*B*		*C*		*C*			*D*
Output :		0		0		1		1		1		0		0		1		1		

Input :		1		0		0		1		1		0		1		0				
State :	*D*		*B*		*C*		*D*		*B*		*D*				*C*					*D*
Output :		0		0		1		0		0		1		1		1				

Input :								0		1										
State :							*D*								*C*					
Output :								1		1										

Up to this point, we have checked every possible transition, except those from D to A and from A to B and C. Since the machine is presently in state C, we must apply a transfer sequence[1] to get to either state D or A. Such sequences can always be found for a strongly connected machine, and require at most $n - 1$ symbols. Furthermore, the transfer sequences should be applied in such a way that they will take the machine through "checked" transitions only. Thus, the only possible transfer sequence in this case is $T(C, D) = 0$, because, as has already been demonstrated, the machine goes from C to D under input symbol 0. The application of a 0 followed by 01 ascertains the transition from D to A and returns the machine back to state D. This sequence provides enough information to verify the transition from A to C under a 1 input symbol. This verification is achieved by inspection of the preceding sequence and observing that C is the 01-successor of D and A is the 0-successor of D. Thus, C is the 1-successor of A.

The last transition that needs to be checked is from state A to B. Since the machine is in state D, a transfer sequence $T(D, A) = 0$ is applied, followed by

[1] Recall that a *transfer sequence* $T(S_i, S_j)$ is the shortest input sequence that takes a machine from state S_i to state S_j.

001. The complete experiment is shown below:

Input :		0		1		0		1		0		1		0		1	
State :	*A*		*B*		*D*		*A*		*C*		*D*		*B*		*C*		*C*
Output :		0		0		1		1		1		0		0		1	

Input :		0		1		0		0		1		1		0		1	
State :	*C*		*D*		*B*		*C*		*D*		*B*		*D*		*A*		*C*
Output :		1		0		0		1		0		0		1		1	

Input :		0		0		1		0		0		0		1		0	
State :	*C*		*D*		*A*		*C*		*D*		*A*		*B*		*D*		*A*
Output :		1		1		1		1		1		0		0		1	

Input :		0		0		1			
State :	*A*		*B*		*C*		*C*		
Output :		0		0		1			

The preset part of the checking experiment thus consists of the above input sequence, whose length is 27 symbols. If the machine at hand responds as shown above then it must be isomorphic to M_4, since it has been shown to contain four states whose responses are identical to the corresponding responses of M_4 and since all state transitions, which have been verified in terms of the behavior exhibited at the beginning of the experiment, are also isomorphic to those of M_4. Clearly, if the machine has not produced the above expected output sequence then it cannot be operating correctly. The location of the fault, however, cannot be determined merely by the above response.

Testing machines that have distinguishing sequences

The procedure can be summarized as follows. A checking experiment starts with a synchronizing sequence or a reset input, so as to maneuver the machine to the desired initial state. The machine is next supplied with an input sequence that causes it to visit each state and display its response to the distinguishing sequence. Finally, the machine is made to go through every state transition and, in each case, the transition is verified by displaying its response to the distinguishing sequence. In practice it is not necessary to display all the responses at the beginning of the experiment. Any response or transition that is verified at a later point in the experiment may be used to determine a state transition at some earlier point.

More precisely, the procedure for constructing checking experiments for machines that have distinguishing sequences is as follows. Let $S_1, S_2, \ldots, S_n$ be the states of machine M, and suppose that X_0 is a distinguishing sequence for this machine. Let Q_i be the state to which M goes, when it is initially in

S_i, as a result of the input sequence X_0. Also, let $T(S_i, S_j)$ denote an input sequence (not necessarily unique) that transfers the machine from state S_i to S_j. Now suppose that M is initially in its starting state S_1. Then, the sequence

$$X_0T(Q_1, S_2)X_0T(Q_2, S_3)X_0T(Q_3, S_4)\cdots X_0T(Q_n, S_1)X_0$$

will serve to take the machine through each of its states and display all the different responses to the distinguishing sequence. For example, starting in S_1, X_0 leaves the machine in Q_1. Then $T(Q_1, S_2)$ transfers the machine to S_2, where X_0 is applied again, leaving the machine in Q_2. The corresponding output sequence clearly displays the response of M to X_0, when initially in either state S_1 or S_2. The machine is similarly led through all its n states and, at each point, the sequence X_0 is applied followed by the transfer sequence $T(Q_i, S_{i+1})$.

At the end of this part of the experiment, the machine receives the sequence $X_0T(Q_n, S_1)$. If it operates correctly, it will be in state S_1. This is verified by applying the distinguishing sequence X_0 to it again. Clearly, if the machine's response to the last X_0 is identical to its response to the first X_0 then it will indeed be in state Q_1 at the end of this part. Thus, the next part of the experiment starts at this point, as the transitions out of state Q_1 are identified.

In the second part of the experiment, we establish various state transitions. To check, for example, the 0-transition out of state S_i, when the machine is initially in some state Q_j, the appropriate sequence is

$$T(Q_j, S_{i-1})X_0T(Q_{i-1}, S_i)0X_0$$

The sequence $T(Q_j, S_{i-1})X_0$ guarantees that the machine indeed goes to state Q_{i-1}, as it did in the previous part of the experiment. The sequence $T(Q_{i-1}, S_i)$ transfers M to state S_i, and then $0X_0$ is applied to cause the 0-transition out of S_i and also to identify it. In a similar manner the machine can be taken through every transition, in each case identifying the transition by means of the response already established in the first part of the experiment. In general, however, to reduce the length of the experiment it is possible to apply the two parts of the experiment simultaneously instead of sequentially.

The method outlined above can be applied to any reduced and strongly connected machine that has at least one distinguishing sequence. The design of checking experiments for machines that do not have any distinguishing sequence is quite complicated, and the resulting experiments are very long. To alleviate this situation, whenever a distinguishing sequence does not exist, extra output terminals can be added to make sure that such a sequence does exist for the augmented machine, as discussed next. Then the above method can be applied to the augmented machine.

Table 13.8 Testing table for M_2

	0/0	0/1	1/0	1/1
A	B	—	D	—
B	A	—	B	—
C	—	D	A	—
D	—	D	C	—
AB	AB	—	BD	—
AC	—	—	AD	—
AD	—	—	CD	—
BC	—	—	AB	—
BD	—	—	BC	—
CD	—	$\boldsymbol{DD}$	AC	—

*13.6 Design of diagnosable machines

A *diagnosable* sequential machine is one that possesses one or more distinguishing sequences and thus permits us to identify uniquely the states of the machine by inspecting its response to such a sequence. In this section, we shall present a method to modify the design of sequential machines in such a way that they will possess special distinguishing sequences for which relatively short checking experiments can be constructed.

The testing graph

The machine M_2 (Table 13.3) does not possess any distinguishing sequence. We shall now show how it may be augmented by an additional output in such a way that the augmented machine will possess several distinguishing sequences.

The state table of M_2 may be rewritten as shown in the upper half of Table 13.8. The column headings consist of all input–output symbol combinations, where the pair I_k/O_l indicates a combination of input symbol I_k and output symbol O_l. The row labels in the upper half of the table are the states of the machine. The entry in column I_k/O_l, row S_i, is the I_k-successor of S_i if this successor is associated with output symbol O_l and is a dash (—) otherwise. For example, the 0-successor of A is B and the corresponding output symbol is 0. Consequently, B is entered in row A under the column 0/0 and a dash is entered in row A under the column 0/1. In a similar manner, the other next-state entries of M_2 are entered in the upper half of the table.

The lower half of the table is derived directly from the upper half. The row labels are all unordered pairs of states, while the table entries are their corresponding successors. If the entries in rows S_i and S_j, column I_k/O_l, of the upper half are S_p and S_q respectively then the entry in row S_iS_j, column I_k/O_l, of the lower half is S_pS_q. For example, since the entries in rows A and

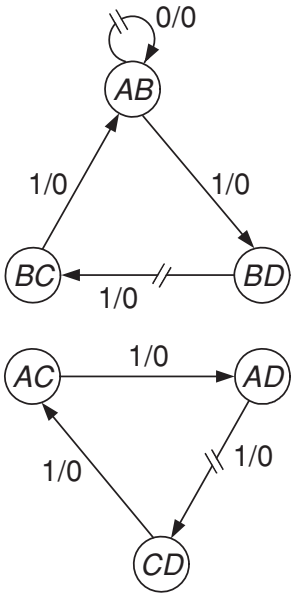


Fig. 13.4 Testing graph for M_2.

B, column $1/0$, are D and B respectively the corresponding entry in row AB, column $1/0$, is BD, and so on. If for some pair of states S_i and S_j, either one or both corresponding entries in some column I_k/O_l are dashes, the corresponding entry in row S_iS_j, column I_k/O_l, is a dash. For example, the entry in row AC, column $0/0$, is a dash, since the entry in row C, column $0/0$, is a dash. The table thus completed is referred to as a *testing table*.

We shall refer to a pair (S_iS_j) as an *uncertainty pair* and to its successor (S_pS_q) as the *implied pair*. Thus, for example, pair (BD) is implied by (AB). An uncertainty pair that does not imply any other pair, so that all the entries in the corresponding row are dashes, can be omitted from the table. Whenever an entry in the testing table consists of a repeated state (e.g., DD in row CD), that entry is given in boldface. Thus the boldface entry DD means that states C and D are *merged*, under input symbol 0, into state D and are indistinguishable by an experiment which starts with a 0 input symbol.

Let us define a directed graph G, which will be called a *testing graph*, in the following way.

1. Corresponding to each row in the lower half of the testing table, there is a vertex in G.
2. If there exists an entry S_pS_q, where $p \neq q$, in row S_iS_j, column I_k/O_l, of the testing table then G has a directed arc leading from the vertex labeled S_iS_j to the vertex labeled S_pS_q. The arc is labeled I_k/O_l. No arc is needed if S_iS_j implies S_pS_p, e.g., DD in row CD.

The testing graph for the machine M_2 is derived directly from the lower half of the testing table and is shown in Fig. 13.4.

Definitely diagnosable machines

A machine M is defined as a *definitely diagnosable machine of order* μ if μ is the least integer such that every sequence of length μ is a distinguishing sequence for M. In other words, a machine is definitely diagnosable if every node at the level μ of the distinguishing tree is associated with a trivial uncertainty vector. The distinguishing tree can thus serve as a tool for recognizing definitely diagnosable machines. In this section, however, we shall derive a different test by means of the testing graph.

Theorem 13.3 *A machine M is definitely diagnosable if and only if its testing graph G is loop-free and no repeated states* (*i.e., boldface entries*) *exist in the testing table.*

Proof If the testing table contains a repeated entry in row S_iS_j, column I_k/O_l, then state S_i cannot be distinguished from state S_j by an experiment that starts with I_k. Thus, if M is definitely diagnosable then its testing table does not contain repeated entries. Now suppose that G is not loop-free. Then, by repeatedly applying the symbols coinciding with the labels of the arcs in the loop, we find an arbitrarily long input sequence that cannot resolve the uncertainty regarding the initial state. Consequently, the machine is not definitely diagnosable. To prove sufficiency, assume that G is loop-free. If M is not definitely diagnosable then there exists an arbitrarily long path in G corresponding to some input sequence X and some pair of states S_iS_j, such that S_i cannot be distinguished from S_j by X. However, since the number of vertices in G cannot exceed $\frac{1}{2}(n-1)n$ (corresponding to the number of distinct pairs of states), arbitrarily long paths in G are possible only if it contains a loop. Thus, the theorem is proved. ◇

The above testing procedure is clearly equivalent to testing by means of the distinguishing tree. In fact, that the graph is loop-free means that no node in the tree is associated with an uncertainty vector whose nonhomogeneous components are also associated with some node in a preceding level. Similarly, if the testing table is free of repeated entries then no node in the tree is associated with an uncertainty vector containing a homogeneous nontrivial component. Hence, every node in the μth level of the tree is associated with a trivial uncertainty vector.

Corollary *Let the testing table of machine M be free of repeated entries, and let G be a loop-free testing graph for M. If the length of the longest path in G is l then* $\mu = l + 1$.

Proof Since G is loop-free, M is definitely diagnosable. Assume that $\mu > l + 1$; then there exists at least one uncertainty pair (S_iS_j) that is transferred, by the application of an input sequence of length $l + 1$, to another pair (S_pS_q). Consequently, there must exist a path, between vertices S_iS_j and S_pS_q in G,

Table 13.9 Machine M_2'

	$NS,\ zz_1$	
PS	$x = 0$	$x = 1$
A	$B, 01$	$D, 00$
B	$A, 00$	$B, 00$
C	$D, 10$	$A, 01$
D	$D, 11$	$C, 01$

whose length is $l + 1$. This contradicts our assumption, and thus μ cannot exceed $l + 1$. The proof that μ cannot be smaller than $l + 1$ is trivial. $\diamond$

We thus arrive at the general result that *if a machine is definitely diagnosable of order* μ, *then* $\mu \geq \frac{1}{2}(n-1)n$. In Problem 13.22, we show that this bound is in fact the least upper bound for μ.

Designing definitely diagnosable machines

In order to obtain a machine M_2' that contains M_2 and possesses a distinguishing sequence, it is necessary to augment M_2 by adding to it an output terminal and assigning different output symbols to selected transitions. We shall, in fact, show that the addition of one output terminal is sufficient to make M_2' definitely diagnosable. The first step toward this end is to assign different output symbols to each transition that may cause a repeated entry in the testing table. In the case of M_2, this is accomplished by assigning the output symbol 10 to the transition from C to D and the output symbol 11 to the transition from D to D. Such an assignment of output values ensures that the testing table of M_2' will be free of repeated entries.

The testing graph of M_2 contains three loops: a self-loop around AB and two other loops, each containing three vertices. Clearly, these loops must be opened if M_2' is to be definitely diagnosable. In general, a loop is opened by the removal of any of its arcs. To remove an arc, it is necessary to assign different output symbols to the next-state entries represented by the vertex to which that arc leads. In other words, an arc leading from the vertex S_iS_j to the vertex S_pS_q is eliminated by assigning different output symbols to the transitions from S_i and S_j to S_p and S_q. For example, the self-loop around AB in Fig. 13.4 is opened by assigning the output symbols 01 and 00, respectively, to the next-state entries B and A in the column $x = 0$. The loop $AB - BD - BC - AB$ can be opened by the removal of the arc from BD to BC. This is achieved by assigning the output symbols 00 and 01 to the next-state entries B and C in rows B and D, column $x = 1$. In a similar manner, we open loop $AC - AD - CD - AC$ by assigning a 00 output symbol to the next-state entry D in row A, column $x = 1$, thus removing the arc from AD to CD. The resulting state table is shown in Table 13.9.

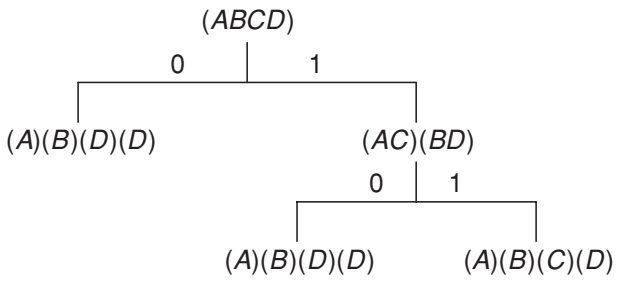


Fig. 13.5 Distinguishing tree for M_2'.

Since the length of a checking experiment is directly proportional to the length of the distinguishing sequence for the machine, we attempt to open all loops while simultaneously minimizing the length of various paths in the graph. In opening the loops in the graph of Fig. 13.4, all the output entries, with the exception of the entry in row C, column $x = 1$, have been assigned new values. The longest path in the loop-free graph is of length 2 and, consequently, the order of the modified machine is $\mu = 3$. This result can, however, be improved by specifying the output entry in row C, column $x = 1$, as 01. This specification actually eliminates the arcs from AC to AD and from BC to AB. As a result, the length of the longest path in the graph is now 1, and M_2' is definitely diagnosable of order 2. The distinguishing tree of machine M_2' is shown in Fig. 13.5.

It is clear that, for any 2^k-state machine, the addition of k output terminals is sufficient to convert it into a definitely diagnosable machine. However, frequently fewer additional output terminals suffice.

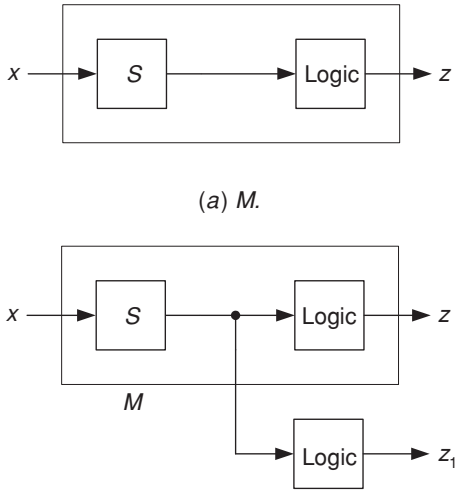
Since the procedure followed in the above example can be applied to any machine, we arrive at the following general result.

- To every reduced machine M there corresponds a definitely diagnosable machine M', which is obtained from M by the addition of one or more output terminals.

The block diagram of the definitely diagnosable machine M' that corresponds to machine M is shown in Fig. 13.6.

A question now arises regarding the purpose of designing definitely diagnosable machines. Evidently, checking experiments can be designed with just one distinguishing sequence. Moreover, even when a machine possesses two or more distinguishing sequences it is not easy to utilize them efficiently and simultaneously in an experiment. The main motivation for designing definitely diagnosable machines and studying their properties is the fact that it is possible to design checking experiments for them. Such experiments are simpler to design for definitely diagnosable machines, since it is possible to cross-check the machine with every sequence of length μ, not with just a single sequence.

Fig. 13.6 Design of a definitely diagnosable machine.



(*a*) *M*.

(*b*) *M* is modified to produce M'. The output z_1 is only used for diagnostic purposes.

13.7 Alternative approaches to the testing of sequential circuits

We saw earlier how finite-state machines can be tested using checking experiments. However, often the test sequences derived by such an approach are quite long. In this section, we shall describe two alternative test generation methods for such machines. The first method also uses a state table; however, the second method uses the sequential circuit implementation of the machine.

State-table-based test generation

This test generation approach uses a *functional fault model*. This fault model assumes that the fault is associated with a state transition in the state table. For example, a *single-state-transition* (SST) fault model assumes that the fault results in the destination state of a state transition becoming corrupted while retaining its correct input/output symbols. Test sequences derived using the SST fault model have been shown to detect a very high percentage of single stuck-at faults in the sequential circuit implementation of the machine.

We shall make the assumption that the SST fault does not increase the number of states in the state table.

We designate each state transition in a machine by the four-tuple

$$< \text{input symbol, source state, destination state, output symbol} > .$$

A state transition can become corrupted if its destination state or output symbol or both are faulty. However, if a test sequence detects a corrupted destination state then it will also detect the corrupted output symbol or both the corrupted destination state and output symbol of that state transition. We can prove this as

follows. In order to detect a corrupt destination state, a test sequence needs to have three parts: an *initialization sequence* that takes the machine to the source state of the state transition in question; the input symbol of the transition to activate the fault; and a *state-pair differentiating sequence* (SPDS) that differentiates between the correct and faulty destination states, i.e., produces different output sequences starting from these states. If the output symbol associated with the state transition is faulty then the initialization sequence and the input symbol that activates the fault together detect the fault. Hence, we can limit our attention to faulty destination states only. We shall derive the three parts of the test sequence from the fault-free state table. Strictly speaking, we should employ both the fault-free and faulty state tables to derive them. However, deriving them from the fault-free state table considerably speeds up the test generation process without much loss in the ability to detect the targeted fault.

An n-state m-transition machine has $m(n-1)$ SST faults. If the machine is large, this number can also be quite large. However, it is possible to use fault collapsing to reduce the number. For each state transition, there are $n-1$ faulty destination states possible. However, we often need to target only a subset of these faulty states. Suppose that the four-tuple $<I_k, S_j, S_i, O_l>$ is corrupted to $<I_k, S_j, S_i', O_l>$ by the SST fault f_1 and to $<I_k, S_j, S_i'', O_l>$ by the SST fault f_2. If we find that the SPDS of S_i and S_i' also differentiates between S_i and S_i'' then fault f_2 dominates fault f_1, and f_2 can be removed from the fault list.

Example Consider the machine M_5 shown in Table 13.10. Since the input symbol $x = 0$ differentiates between states A and B as well as A and C, $SPDS(A, B) = SPDS(A, C) = 0$. Similarly, $SPDS(B, C) = 1$. Next, consider the state transition $<1, C, A, 0>$. Its destination state A can be corrupted in three ways to give $<1, C, B, 0>$, $<1, C, C, 0>$, or $<1, C, D, 0>$. However, since $SPDS(A, B)$ is also $SPDS(A, C)$, the first two of these faulty transitions can be collapsed into just the first one.

Table 13.10 Machine M_5

	NS, z	
PS	$x = 0$	$x = 1$
A	$C, 0$	$C, 0$
B	$C, 1$	$B, 1$
C	$D, 1$	$A, 0$
D	$A, 0$	$B, 1$

Another reasonable fault-collapsing heuristic, which does not reduce the SST fault coverage much, is the following. If two state transitions have an identical source state, destination state, and output label then they are collapsed into a single transition. For example, in the machine in Table 13.10 the two

transitions from state A satisfy this condition. Hence, only one of the these faulty transitions needs to be considered, not both.

Before test generation starts, we first compute transfer sequences between every pair of states. Then, we compute the relevant SPDSs. Test generation consists of the following three steps.

1. *Initialization* In this step the machine is brought from the current state to the source state of the faulty transition using an appropriate transfer sequence.
2. *Excitation* In this step the faulty transition is executed.
3. *State differentiation* In this step the corresponding SPDS is applied to differentiate between the good and faulty states.

Example Consider an SST fault that corrupts the transition $<0, D, A, 0>$ to $<0, D, B, 0>$ in machine M_5. To derive a test sequence for this SST, we first need $T(A, D) = 00$ (10 is also a valid transfer sequence). Then the activation vector $x = 0$ is applied. Finally, $SPDS(A, B) = 0$ is applied. Hence, one possible test sequence is 0000.

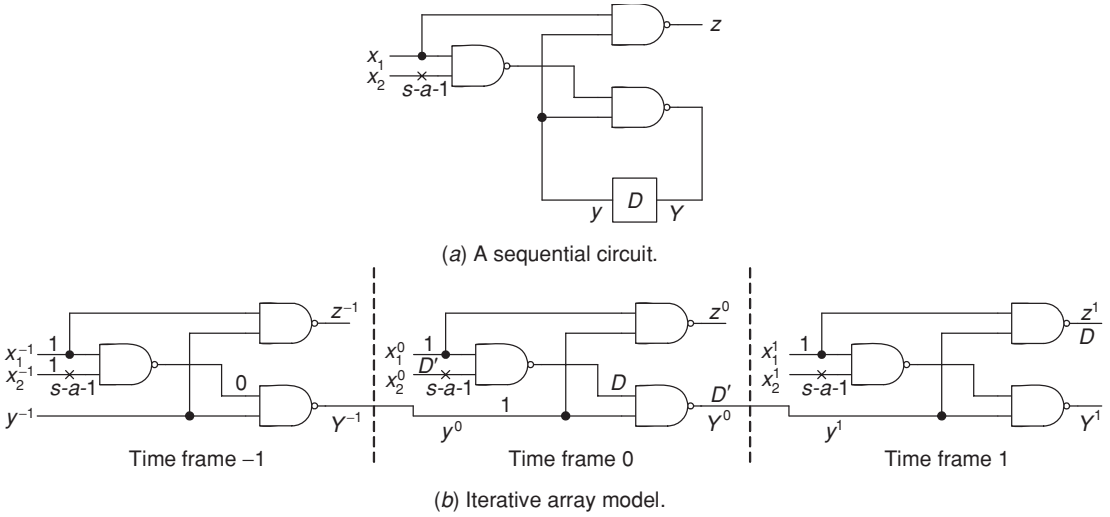
Sequential circuit based test generation

In this subsection, we shall show how test generation can be performed for sequential circuits with the help of the iterative array model. An iterative-array model of a sequential circuit was presented in Chapter 9. This approach generates a test sequence to activate the fault and propagate its effect to a circuit output by finding a sensitized path through multiple time frames.

Since we are targeting a faulty sequential circuit, we shall assume that the initial state of the circuits is unknown.

Extended *D*-algorithm

In Chapter 8, we discussed the D-algorithm, which can be used to generate test vectors for faults in combinational circuits. It is possible to extend the D-algorithm to generate test sequences for sequential circuits. We can target a fault in some time frame, say time frame 0, in the iterative array model and use the D-algorithm to generate a test vector for it. If a D or D' propagates to a circuit output, no further error propagation is required. However, if it only propagates to the next-state lines, we need to add a new time frame as the next time frame, labeled time frame 1, to try to propagate the error signal further. This process is repeated until the error signal reaches some circuit output. If the test vector contains assignments of specific logic values to any present state lines in time frame 0, we add a new time frame as the previous time frame, labeled time frame -1. We then try to justify (trace) the current state backwards through the previous time frame. This process of line justification (Section 8.2) is repeated until no particular logic values are required at the present state lines.



(*a*) A sequential circuit.

(*b*) Iterative array model.

Fig. 13.7 Application of the extended D-algorithm.

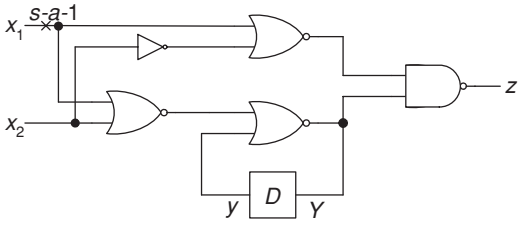
Example Consider the sequential circuit shown in Fig. 13.7*a* and its iterative array model in Fig. 13.7*b*. The signals are superscripted with i in time frame i. Suppose the target stuck-at fault is s-a-1 on input x_2. This targeted stuck-at fault has to be included in every time frame. First, let us consider time frame 0. After applying the D-algorithm to the stuck-at fault in this time frame, the error signal D' propagates to the next-state line Y and the value 1 needs to be justified at the present state line y. To propagate the error further, time frame 1 is added to the right. The error signal can now be propagated to the circuit output z in this time frame. Therefore, we need to justify the value at y in time frame 0. A time frame -1 is added to the left of time frame 0 for this purpose. The signal $x_1 = x_2 = 1$ is needed at the input of this time frame to obtain $y = 1$ in time frame 0. Since the stuck-at fault is present in each time frame, we need to make sure that the fault-free and stuck-at values are the same in this state justification step. Since no value was assigned to line y in time frame -1, there is no need to add any further time frames to the left. We thus arrive at the test sequence for the above fault, consisting of vectors at inputs (x_1, x_2), as $\{(1, 1), (1, 0), (1, \phi)\}$.

The nine-valued logic

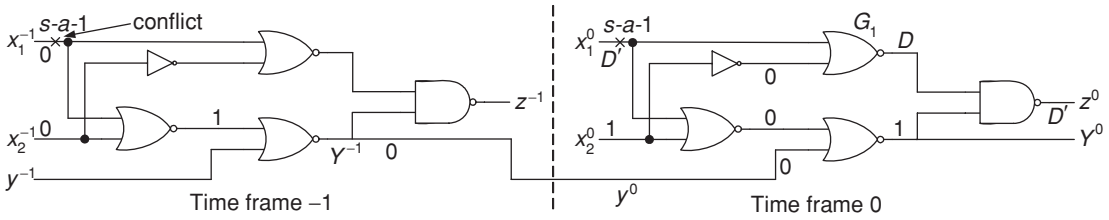
Although the above extension of the D-algorithm is straightforward, the five-valued logic $\{0, 1, \phi, D, D'\}$ used in the D-algorithm is not adequate for sequential circuits because it overspecifies the value requirements at some lines in the circuit. This may prevent the test generator from obtaining a test sequence even when one exists. This problem can be tackled by using a nine-valued logic instead. This logic accounts for the effects of the fault in each time frame correctly. The nine values in this logic each represent an ordered

pair from the ternary values 0, 1, and ϕ. The first value of the pair represents the ternary value of the fault-free circuit and the second value represents the ternary value of the faulty circuit. Hence, the nine ordered pairs are $0/0$, $0/1$, $0/\phi$, $1/0$, $1/1$, $1/\phi$, $\phi/0$, $\phi/1$, and ϕ/ϕ.

We next illustrate through an example why nine-valued logic succeeds where the extended D-algorithm may not.



(a) A sequential circuit.



(b) Application of the extended D-algorithm.

Fig. 13.8 Test generation with five-valued logic.

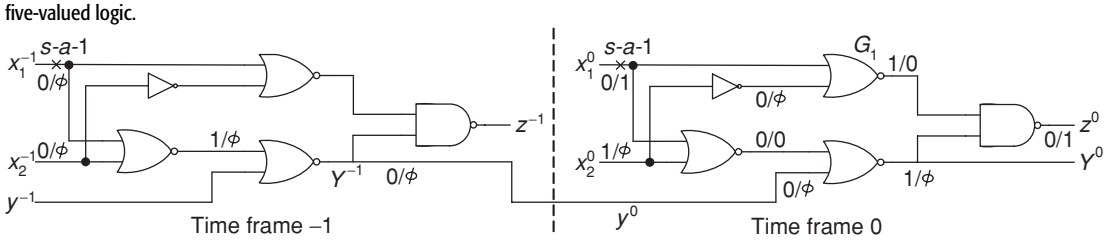


Fig. 13.9 Test generation with nine-valued logic.

Example Consider the sequential circuit shown in Fig. 13.8a. Application of the extended D-algorithm to this circuit is illustrated in Fig. 13.8b. Since, with the shown logic values in time frame 0, the error signal D' gets propagated to the circuit output, there is no need to add time frame 1 to the right. However, we need to add time frame -1 to the left in order to justify the value 0 required at y^0. Justifying the values in time frame -1 results in a conflict at x_1^{-1}, on which a 0 is required whereas it has an s-a-1 fault present. Thus the algorithm concludes that no two-vector test sequence exists to detect this fault. Note that if a 1 had been placed at y^{-1} to justify a 0 at Y^{-1}, then we would need to add time frame -2 to the left.

Next, consider the application of nine-valued logic to this circuit, as shown in Fig. 13.9. In order to propagate the error from x_1 to the output of gate G_1 in time frame 0, the other input of G_1 must have a 0 for the fault-free circuit but does not require any particular value for the faulty circuit. This is denoted as $0/\phi$. Eventually, we require $0/\phi$ at the line y^0. Owing to this relaxed requirement, there is no conflict at x_1^{-1}. The corresponding test sequence for this fault is thus $\{(0,0), (0,1)\}$.

13.8 Design for testability

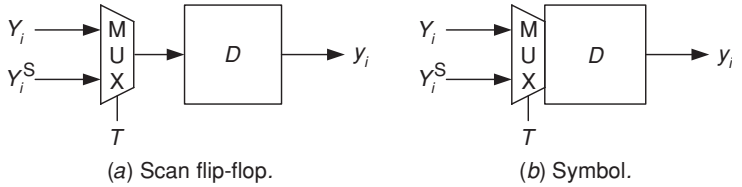
Since it is difficult to control the present-state lines and observe the next-state lines of a sequential circuit, sequential test generation generally does not lead to a high fault coverage. When certain design features are added to a circuit to make it easier to derive tests or test sequences for the circuit, the corresponding approach is called *design for testability*.

Scan design

A popular design-for-testability approach for sequential circuits is called scan design. In *scan design* there are two modes of operation, *normal* and *test*. In the normal mode, the circuit exhibits its original input–output behavior. However, in the test mode the flip-flops of the circuit are chained into a shift register. If all the flip-flops of the circuit are so chained, the circuit is said to be a *full-scan design*. If a fraction, but not all, of the flip-flops are so chained, the resultant circuit is said to be a *partial-scan design*.

Since a flip-flop may have two sources of inputs, one corresponding to its normal mode of operation and another corresponding to its test mode, a special flip-flop, called a *scan flip-flop*, is needed. Such a scan flip-flop essentially has a 2-to-1 multiplexer at its input, as shown in Fig. 13.10*a*. The ith D flip-flop has a normal-mode input Y_i and a test-mode input Y_i^S (where the superscipt S denotes "scan"). When the *mode-select signal* T is 0, the upper input of the multiplexer is selected and this corresponds to the normal mode. However, when $T = 1$ the lower input is selected and this corresponds to the test mode. We shall, henceforth, use the compact symbol shown in Fig. 13.10*b*.

We are now in a position to analyze the scan chain shown in Fig. 13.11. An extra input, called the scan input (labeled *ScanIn*) and an extra output, called



(*a*) Scan flip-flop. (*b*) Symbol.

Fig. 13.10 A flip-flop with an input multiplexer.

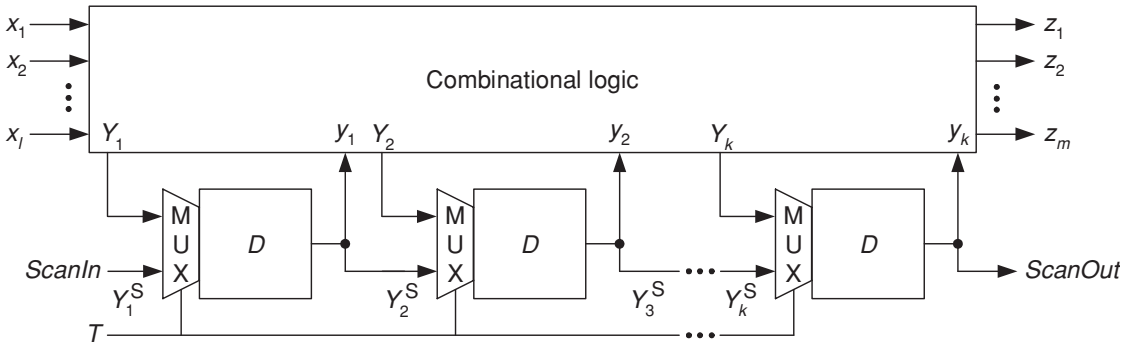


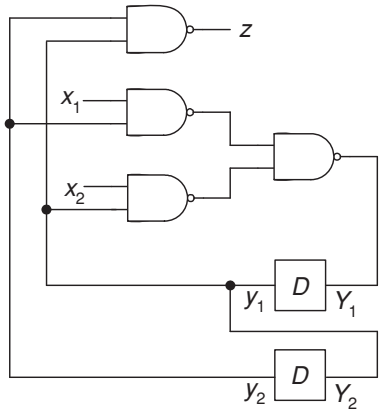
Fig. 13.11 A scan chain.

the scan output (labeled *ScanOut*), are added to the circuit. When $T = 0$, the next-state value at line Y_i, $1 \leq i \leq k$, gets transferred to the present-state line y_i after the flip-flop is clocked, as one would expect during the normal operation of a sequential circuit. However, when $T = 1$ the value at *ScanIn* is transferred to the output of the first flip-flop after clocking, the output of the first flip-flop gets transferred to the output of the second flip-flop, and so on. In other words, the value at $Y_i^S = y_{i-1}$ gets transferred to y_i. The value at y_k also gets propagated to *ScanOut*.

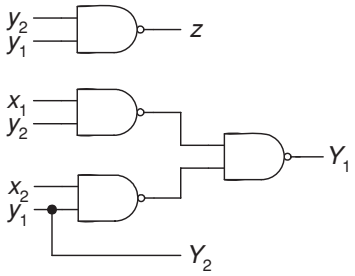
Testing of circuits using scan design

The scan chain enables any state of the sequential circuit to be scanned into the flip-flops in the test mode, essentially making the flip-flops fully controllable. After applying the test to the circuit, the next-state values can be captured in the flip-flops in the normal mode. Then these values can be shifted out through *ScanOut* in the test mode, thus also making the flip-flops fully observable. This reduces the sequential test generation problem to that of test generation for the combinational logic of the circuit. This logic has $x_1, x_2, \ldots, x_l, y_1, y_2, \ldots, y_k$ as primary inputs and $z_1, z_2, \ldots, z_m, Y_1, Y_2, \ldots, Y_k$ as circuit outputs. Thus, the test vectors for such a circuit will have $l + k$ bits and the resulting output response will have $m + k$ bits. The first l input bits are said to constitute the *primary input part* of the vector and last k input bits its *state part*. A test set can be obtained for such a circuit using any combinational test generation algorithm, e.g., the D-algorithm presented in Chapter 8 if stuck-at faults are the targeted faults.

Example Consider the sequential circuit in Fig. 13.12*a*. Its combinational logic is shown in Fig. 13.12*b*. Readers can check that the test set shown in Fig. 13.12*c* detects all single stuck-at faults in this combinational logic. The first two bits of each of the test vectors in this set denote the primary input part and the last two bits its state part.



(*a*) Sequential circuit.



(*b*) Combinational logic.

x_1	x_2	y_1	y_2
1	0	0	1
0	1	1	0
1	1	0	0
0	0	1	1

(*c*) Stuck-at fault test set.

Fig. 13.12 Testing of scan designs.

To apply the test set derived for the combinational logic to the sequential circuit, the following procedure can be followed.

1. Make $T = 1$ to set the sequential circuit into test mode.
2. Scan in the state part of the vector through the *ScanIn* input in the next k clock cycles. In these cycles, the primary inputs can be fed arbitrary values.
3. Apply the primary input part of the vector to the primary inputs. At this point, all the $l + k$ bits of the test vector have been applied to the combinational logic. After allowing the combinational logic to settle down, observe the output response at circuit outputs $z_1, z_2, \ldots, z_m$.
4. Make $T = 0$ to set the circuit into normal mode.
5. Apply a clock pulse. This results in the values on the next state lines, Y_1, $Y_2, \ldots, Y_k$, being latched in the k flip-flops.
6. Make $T = 1$ and observe the values captured in the flip-flops by scanning them out through *ScanOut* while repeating this procedure for the next test vector.

The flip-flops are themselves tested beforehand by shifting through them a sequence of 1's and then a sequence of 0's to make sure that both a 1 and a 0 can be shifted through each flip-flop.

Suppose that there are n test vectors in the test set. A total of k clock cycles are required to scan-in the state part, one cycle to capture the state response, and $k - 1$ clock cycles to scan-out the captured state. Since the state part of the next test vector is scanned-in at the same time as the captured state for the previous vector is being scanned out, the total number of clock cycles needed to apply the complete test set is $n(k + 1) + k - 1$.

Example For the test set in Fig. 13.12*c*, $n = 4$ and $k = 2$. Thus, a total of 13 clock cycles is required for it.

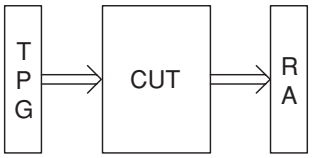


Fig. 13.13 A circuit with BIST.

13.9 Built-in self-test (BIST)

The BIST approach allows the circuit to test itself. This requires that some extra circuitry be integrated on-chip. It reduces the need for expensive automatic test equipment. It allows the test vectors to be applied to the circuit under test (CUT) at the normal clock rate. This is called *at-speed testing* and has been found useful for detecting delay faults. A chip with BIST can also be tested in the field, which enhances the reliability of the system.

A CUT that incorporates BIST is shown in Fig. 13.13. It contains a test pattern generator (TPG), CUT, and response analyzer (RA). The TPG generates pseudo-random test sequences and applies them to the CUT. The RA compresses the output response of the CUT into a vector called the *signature*. When there is no fault present in the CUT, the corresponding compressed response is called the *golden signature*. When a fault is present, it is highly likely that the compressed response will not match the golden signature, thus indicating the presence of a fault.

Test pattern generator

The TPG usually comprises a linear feedback shift register (LFSR). An LFSR consists of D flip-flops and XOR gates. It belongs to the class of linear sequential machines which will be discussed in detail in Chapter 15.

A k-stage LFSR is shown in Fig. 13.14 (the number of stages refers to the number of flip-flops present). In it, the output y_1 of the last flip-flop is fed back to a subset of the flip-flops determined by whether the corresponding b_j, $1 \leq j \leq k$, is 0 or 1. The presence (absence) of the feedback is indicated by $b_j = 1$ ($b_j = 0$). An LFSR is often described by a feedback polynomial:

$$p(x) = x^k + b_1 x^{k-1} + \cdots + b_{k-1} x + b_k.$$

Such a polynomial is said to have *degree* k.

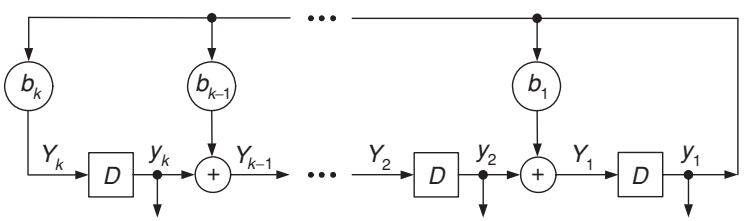


Fig. 13.14 A k-stage LFSR.

Example Consider the three-stage LFSR shown in Fig. 13.15. Its feedback polynomial is $p(x) = x^3 + x^2 + 1$. Note that in this case $b_3 = b_1 = 1$ and $b_2 = 0$.

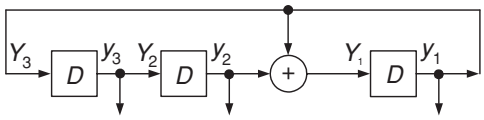


Fig. 13.15 An example of a three-stage LFSR.

A feedback polynomial is said to be *primitive* if the state diagram of the corresponding k-stage LFSR consists of two loops, a trivial loop with the all-0 state and a nontrivial loop with the remaining $2^k - 1$ states. The outputs of the k flip-flops can be directly fed to the inputs of a k-input CUT. The output patterns generated by such an LFSR are known to have very good randomness properties and hence are very useful for obtaining a high coverage of faults in the CUT.

Example For the three-stage LFSR shown in Fig. 13.15, the state diagram is shown in Fig. 13.16. Thus its feedback polynomial $p(x) = x^3 + x^2 + 1$ is primitive.

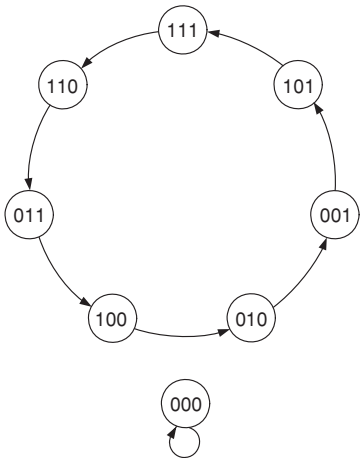


Fig. 13.16 State diagram of the LFSR in Fig. 13.15.

A list of primitive polynomials for various values of k is known. As an example, readers can verify that $p_1(x) = x^4 + x + 1$ is a primitive polynomial but that $p_2(x) = x^4 + x^2 + 1$ is not.

Usually, LFSRs based on primitive polynomials find use in BIST. Test pattern generation can start with any state in the nontrivial loop of such an LFSR. The initial state is called the *seed*. Clocking of the LFSR causes it to transition

from the seed to the next state, and so on. For example, in the state diagram in Fig. 13.16, if the seed is state 001 then the next state will be 101 and then 111, and so on. These patterns can be fed to a three-input CUT in order to test it. It is possible, however, that many patterns from this test sequence are not needed to detect any targeted faults in the CUT. Thus, if we started from different seeds and applied a few test patterns from each, we could shorten the time it takes to test the CUT. This process is called *LFSR re-seeding*.

Example Consider the circuit shown in Fig. 13.17. A possible test set for detecting all single stuck-at faults in this circuit is $(x_3, x_2, x_1) = \{(1, 0, 1), (1, 1, 1), (1, 0, 0), (0, 1, 0)\}$. Suppose that the LFSR shown in Fig. 13.15 is used to test it, with y_i connected to the input x_i, $1 \leq i \leq 3$. From the state diagram in Fig. 13.16, we can see that testing can be accomplished by applying two patterns starting with the seed (1, 0, 1) and two additional patterns starting with the seed (1, 0, 0). The two seeds can be stored on-chip and fed to the LFSR when needed. Thus, we see that four clock cycles are needed to test this circuit, which is the minimum possible. However, if only one seed were used, say (1, 0, 1), then we would have to cycle through six patterns from (1, 0, 1) to (0, 1, 0), for a total of six clock cycles.

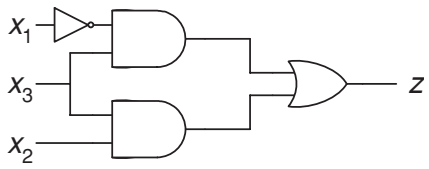


Fig. 13.17 Re-seeding example.

Response analyzer

For a k-output CUT to which v test patterns have been applied by the TPG, we need to analyze the kv output bits to see if any bit is erroneous, thus indicating the presence of a fault in the CUT. To do this, we would need to store the fault-free values of these bits and do a bit-by-bit comparison with the response obtained from the CUT. Since this can be quite expensive in terms of space and time, output responses are usually compressed into a signature and compared with the golden signature that would be obtained if no faults were present in the CUT. However, it is possible that, even if erroneous bits are present in the response, its signature is the same as the golden signature. This is called *aliasing*. This will lead us to declare a faulty circuit to be fault-free, which is obviously a scenario we would like to avoid. Luckily, the aliasing probability of an RA is typically extremely small.

A commonly used RA is the multiple-input signature register (MISR), which is obtained by modifying an LFSR, as shown in Fig. 13.18. The k outputs of the CUT, $z_1, z_2, \ldots, z_k$, are connected to the k-stage MISR as shown. When

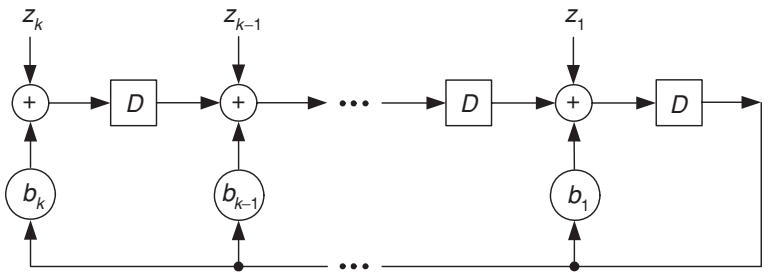


Fig. 13.18 A multiple-input signature register.

the CUT is being tested, in each cycle a k-bit response is fed to the MISR, leading it to a new state. When the final k-bit response is fed to the MISR, the state it enters is the signature in which we are interested. It has been shown that the aliasing probability of such a MISR is close to $1/2^k$ (note that this is independent of the CUT under test). For reasonable values of k, such as $k = 32$, this probability is negligible.

Appendix 13.1 Bounds on the length of synchronizing sequences

We shall next establish a range of values for the length of a synchronizing sequence and show that the value of the least upper bound on the length must be in this range.

Theorem 13.4 *If an n-state machine has a synchronizing sequence, or sequences, then it has one such sequence whose length is at most* $\frac{1}{6}n(n+1)(n-1)$.

Proof A necessary condition for a machine to have a synchronizing sequence is that, under at least one input symbol I_k, the I_k-successors of some two states S_i, S_j will be identical. The synchronization of a machine, whose initial state is unknown, into some state S_c can be accomplished by applying I_k to the machine in such a that way if it is in either S_i or S_j then it will go to the common successor; next, a sequence that transfers another pair of states S_p, S_q into S_i, S_j is applied, and after that I_k is again applied to the machine to take it into the common successor, and so on. This process actually reduces the initial uncertainty $(S_1 S_2 \cdots S_n)$ to the singleton uncertainty (S_c).

Suppose now that $k-1$ states have already been taken out of the uncertainty, which presently consists of $n-k+1$ states. We wish to obtain an upper bound on the length of the sequence needed to reduce the uncertainty by another state, that is, to reduce it to $n-k$ states. Suppose also that S_u and S_v are the states that will now be taken by this sequence into a common successor. The present uncertainty U thus consists of S_u, S_v, and the remaining $n-k-1$ states. The length of the required sequence depends on the number of pairs of states through which $S_u S_v$ passes before reaching the common successor. This number will be maximized if $S_u S_v$ does not pass through any other pair of states contained in the remaining $n-k-1$ states of the uncertainty (because

in such a case we could use that pair of states to reduce the uncertainty). For the same reason, $S_u S_v$ should not pass through any pair of states contained in the successors of these $n-k-1$ states.

Thus the length of the sequence to be obtained will be maximized if all the uncertainty successors of U contain the same $n-k-1$ states and only $S_u S_v$ passes through various pairs of states. The successors of $S_u S_v$ may be any pair of states not contained in these $n-k-1$ states. Since there are $n-(n-k-1)=k+1$ such states, there are $\frac{1}{2}k(k+1)$ pairs of possible successors to $S_u S_v$. Consequently, at most $\frac{1}{2}k(k+1)$ (which is equal to $1+2+3+\cdots+k$) input symbols are needed to take out the kth state from the uncertainty.

To reduce the initial uncertainty $(S_1 S_2 \cdots S_n)$ to a singleton uncertainty, a sequence of length $1+(1+2)+(1+2+3)+\cdots+(1+2+3+\cdots+n-1)=\sum_{k=2}^{n}\frac{1}{2}k(k-1)$ is needed. Since $\frac{1}{2}k(k-1)=0$ for $k=1$, we can take the sum from 1 to n, i.e.,

$$\begin{aligned}\frac{1}{2}\sum_{k=1}^{n}k(k-1) &= \frac{1}{2}\sum_{k=1}^{n}k^2-\frac{1}{2}\sum_{k=1}^{n}k\\ &= \frac{1}{2}\left[\frac{n(n+1)(2n+1)}{6}-\frac{3n(n+1)}{6}\right]\\ &= \frac{n(n+1)(n-1)}{6}.\end{aligned}$$

◇

Theorem 13.4 thus establishes an upper bound on the length of synchronizing sequences, which is lower by a constant factor than that in Section 13.2.

Theorem 13.5 *For every n, there exists an n-state machine that has a synchronizing sequence of length* $(n-1)^2$.

Proof A machine that satisfies the theorem is given in Table 13.11. The proof that the shortest synchronizing sequence for this machine is of the form $0(1^{n-1}0)^{n-2}$ is left to the reader as a (nontrivial) exercise. Note that the proof must consist of two parts: first, the proof that the above is indeed a synchronizing

Table 13.11 A machine with a synchronizing sequence of length $(n-1)^2$

	NS	
PS	$x=0$	$x=1$
S_1	S_1	S_n
S_2	S_1	S_1
S_3	S_3	S_2
⋮	⋮	⋮
S_k	S_k	S_{k-1}
⋮	⋮	⋮
S_{n-1}	S_{n-1}	S_{n-2}
S_n	S_n	S_{n-1}

sequence, and second a demonstration that it is the shortest synchronizing sequence.

The length of the subsequence within the parentheses is n, since it consists of $n-1$ 1's followed by a 0. There are $n-2$ such subsequences, preceded by a single 0. Hence, the total length is $1+(n-2)n = n^2-2n+1 = (n-1)^2$. $\diamond$

Example A machine that illustrates Theorem 13.5 for $n = 5$ is shown in Fig. 13.19*a*. The corresponding path in the synchronizing tree, which leads to the singleton uncertainty, is given in Fig. 13.19*b*.

	NS	
PS	$x=0$	$x=1$
S_1	S_1	S_5
S_2	S_1	S_1
S_3	S_3	S_2
S_4	S_4	S_3
S_5	S_5	S_4

(*a*) Machine M_6

$(S_1S_2S_3S_4S_5)$
↓ 0
$(S_1S_3S_4S_5)$
↓ 1
$(S_2S_3S_4S_5)$
↓ 1
$(S_1S_2S_3S_4)$
↓ 1
$(S_1S_2S_3S_5)$
↓ 1
$(S_1S_2S_4S_5)$
↓ 0
$(S_1S_4S_5)$
↓ 1
$(S_3S_4S_5)$
↓ 1
$(S_2S_3S_4)$
↓ 1
$(S_1S_2S_3)$
↓ 1
$(S_1S_2S_5)$
↓ 0
(S_1S_5)
↓ 1
(S_4S_5)
↓ 1
(S_3S_4)
↓ 1
(S_2S_3)
↓ 1
(S_1S_2)
↓ 0
(S_1)

(*b*) Shortest synchronizing sequence for M_6

Fig. 13.19 Demonstrating Theorem 13.5 for $n = 5$.

Combining the results in Theorems 13.4 and 13.5, we obtain the following corollary.

Corollary *The least upper bound L on the length of synchronizing sequences is bounded by* $(n-1)^2 \leq L \leq \frac{1}{6}n(n+1)(n-1)$.

Appendix 13.2 A bound on the length of distinguishing sequences

Next, we prove that the length of the distinguishing tree is bounded and, consequently, the construction of such a tree is a finite process.

Theorem 13.6 *If a preset distinguishing sequence for an n-state machine M exists then its length is at most* $(n-1)n^n$.

Proof Let the uncertainty vector at some level in the distinguishing tree consist of m components whose sizes are $k_1, k_2, \ldots, k_m$. Clearly, the sum of the sizes of all the components must be equal to n; i.e., $k_1 + k_2 + \cdots + k_m = n$. Let the numbers $k_1, k_2, \ldots, k_m$ be subsets in a partition μ such that $\mu = \{k_1, k_2, \ldots, k_m\}$. Clearly, μ defines the *size distribution* of the components in the uncertainty vector. The number of different uncertainty vectors with the same size distribution μ is equal to $n^{k_1} n^{k_2} \cdots n^{k_m} = n^n$.

Consider now a path in the tree leading from the initial uncertainty vector to a trivial uncertainty vector. Let U_1 and U_2 be uncertainty vectors along this path, with corresponding partitions μ_1 and μ_2. Clearly, if U_2 is a successor of U_1 then the size distribution of U_2 is either equal to that of U_1 or is a refinement of that of U_1; i.e., $\mu_1 \geq \mu_2$. Also, since the initial uncertainty vector contains n states, there are at most $n-1$ possible refinements of partitions along the path leading to the distinguishing sequence. Accordingly, the length of this path is $L \leq (n-1)n^n$. ◇

The above bound is not necessarily the least upper bound.

Notes and references

The study of machine behavior from terminal experiments was first introduced by Moore [13] in 1956. He established the notions of homing, synchronizing, and distinguishing experiments and derived bounds on their lengths. Moore's ideas were further developed by Gill [5], who simplified the search for the homing and distinguishing sequences, Ginsburg [6], Hibbard [8], and Kohavi and Winograd [12]. The material on checking experiments is taken from Hennie [7], Kohavi and Lavallee [10], Kohavi and Kohavi [9], and Kohavi *et al.* [11]. State-table-based test generation using a functional fault model was presented by Cheng and Jou [2]. A survey of sequential test generation methods was presented by Cheng [3]. Sequential test generation based on nine-valued logic was first presented by Muth [14]. Scan design was first discussed by Williams and Angell [15]. A level-sensitive scan design, which is quite influential, was discussed by Eichelberger and Williams [4]. A more detailed description of BIST techniques can be found in the book by Bardell, McAnney, and Savir [1].

[1] Bardell, P. H., W. H. McAnney, and J. Savir: *Built-in Test for VLSI: Pseudorandom Techniques*, John Wiley & Sons, 1987.
[2] Cheng, K.-T., and J.-Y. Jou: "A functional fault model for finite state machines," *IEEE Trans. Computer-Aided Design*, vol. 11, no. 9, pp. 1065–1073, September 1992.
[3] Cheng, K.-T.: "Gate-level test generation for sequential circuits: a survey," *ACM Trans. Design Automation of Electronic Systems*, vol. 1, no. 3, pp. 405–442, 1996.
[4] Eichelberger, E. B., and T. W. Williams: "A logic design structure for design for testability," in *Proc. Design Automation Conf.*, pp. 462–468, June 1977.
[5] Gill, A.: "State-identification experiments in finite automata," *Information and Control*, vol. 4, pp. 132–154, 1961.
[6] Ginsburg, S.: "On the length of the smallest uniform experiment which distinguishes the terminal states of a machine," *J. Assoc. Computing Machinery*, vol. 5, pp. 266–280, July 1958.
[7] Hennie, F. C.: "Fault detecting experiments for sequential circuits," in *Proc. Fifth Ann. Symp. Switching Circuit Theory and Logical Design*, pp. 95–110, November 1964.
[8] Hibbard, T. N.: "Least upper bounds on minimal terminal state experiments for two classes of sequential machines," *J. Assoc. Computing Machinery*, vol. 8, pp. 601–612, October 1961.
[9] Kohavi, I., and Z. Kohavi: "Variable-length distinguishing sequences and their application to the design of fault-detection experiments," *IEEE Trans. Computers*, vol. C-17, pp. 792–795, August 1968.
[10] Kohavi, Z., and P. Lavallee: "Design of sequential machines with fault-detection capabilities," *IEEE Trans. Electron. Computers*, vol. EC-16, pp. 473–484, August 1967.
[11] Kohavi, Z., J. A. Rivierre, and I. Kohavi: "Checking experiments for sequential machines," *Information Sciences*, vol. 7, no. 1, pp. 11–28, January 1974.
[12] Kohavi, Z., and J. Winograd: "Establishing bounds concerning finite automata," *J. Computer & System Sciences*, vol. 7, no. 3, pp. 288–299, June 1973.
[13] Moore, E. F.: "Gedanken-experiments on sequential machines," pp. 129–153, *Automata Studies*, Princeton University Press, 1956.
[14] Muth, P.: "A nine-valued circuit model for test generation," *IEEE Trans. Computers*, vol. C-25, no. 6, pp. 630–636, June 1976.
[15] Williams, M., and J. Angell: "Enhancing testability of large-scale integrated circuits via test points and additional logic," *IEEE Trans. Computers*, vol. C-32, pp. 46–60, 1973.

Problems

Problem 13.1. For each machine shown in Table P13.1:
(a) find the shortest homing sequences;
(b) determine whether synchronizing sequences exist, and if any do exist, find the shortest ones.

Table P13.1

	NS, z	
PS	$x=0$	$x=1$
A	$A,1$	$E,0$
B	$A,0$	$C,0$
C	$B,0$	$D,1$
D	$C,1$	$C,0$
E	$C,0$	$D,0$

M_1

	NS, z	
PS	$x=0$	$x=1$
A	$B,0$	$A,0$
B	$B,1$	$C,1$
C	$A,1$	$D,0$
D	$C,0$	$A,1$

M_2

	NS, z	
PS	$x=0$	$x=1$
A	$C,0$	$D,1$
B	$C,0$	$A,1$
C	$A,1$	$B,0$
D	$B,0$	$C,1$

M_3

Problem 13.2. It is necessary to synchronize the machine of Table P13.2 to state A with a minimum number of input symbols. Devise such a procedure.

Table P13.2

	NS, z	
PS	$x=0$	$x=1$
A	$C,1$	$E,1$
B	$A,0$	$D,1$
C	$E,0$	$D,1$
D	$F,1$	$A,1$
E	$B,1$	$F,0$
F	$B,1$	$C,1$

Problem 13.3. You are presented with a machine that is known to be described by one of the two state tables shown in Table P13.3. No information is available regarding the initial state of the machine. Devise a procedure for identifying the machine, and find all minimal preset experiments that can perform this task.

Hint: Construct a machine which is the *direct sum* of the two machines.

Table P13.3

	NS, z	
PS	$x=0$	$x=1$
A	$A,0$	$B,0$
B	$C,0$	$A,0$
C	$A,1$	$B,0$

	NS, z	
PS	$x=0$	$x=1$
D	$E,0$	$F,1$
E	$F,0$	$D,0$
F	$E,0$	$F,0$

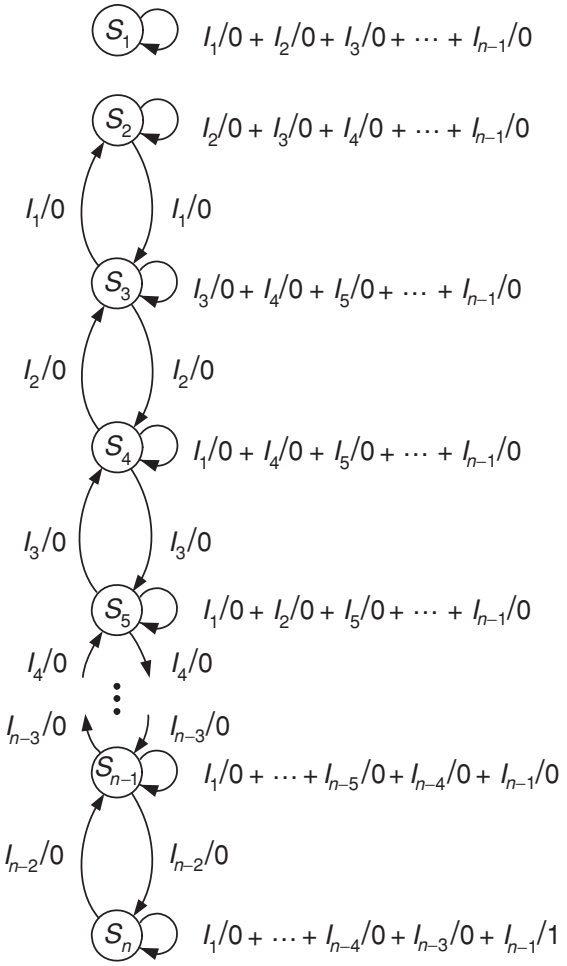
Problem 13.4. Find the shortest homing sequence for the machine shown in Table P13.4. (Note that this machine is a special case, $n=4$, of the machine of Fig. P13.5.)

Table P13.4

	NS, z		
PS	I_1	I_2	I_3
S_1	$S_1, 0$	$S_1, 0$	$S_1, 0$
S_2	$S_3, 0$	$S_2, 0$	$S_2, 0$
S_3	$S_2, 0$	$S_4, 0$	$S_3, 0$
S_4	$S_4, 0$	$S_3, 0$	$S_4, 1$

Problem 13.5. It can be shown that every n-state machine has a preset homing sequence whose length does not exceed $\frac{1}{2}(n-1)n$. By referring to Fig. P13.5, prove that this bound cannot be lowered; i.e., there exists a class of machines the length of whose homing sequences is precisely $\frac{1}{2}(n-1)n$.

Fig. P13.5



Problem 13.6

(a) Find a single sequence of 0's and 1's that can serve as a homing sequence for all reduced and strongly connected three-state machines whose input symbols are 0 and 1.

(b) Can you generalize the result of part (a) to n-state machines? Show a bound on the length of such sequences.

Problem 13.7. Prove that, in a reduced n-state machine, every set of $n - k$ states ($n - 2 \geq k \geq 0$) contains at least one pair of states that is distinguishable by an experiment of length $k + 1$.

Problem 13.8. It is necessary to determine the final state of the machine shown in Table P13.8 when the initial state is unknown and only output sequences from the machine are available to the experimenter; that is, no information regarding the input to the machine is available.

(a) Devise a procedure to determine whether a specific output sequence can be used to identify the final state of the machine.

(b) Find a reduced standard-form state table that accepts precisely those output sequences which can be used to identify the final state of the machine. Use the state names A, B, etc.

Table P13.8

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 0$	$C, 0$
B	$A, 0$	$D, 1$
C	$D, 1$	$B, 0$
D	$A, 1$	$D, 1$

Problem 13.9. For each of the machines shown in Table P13.9, determine whether preset distinguishing sequences exist, and if any do exist then find the shortest ones.

Table P13.9

	NS, z	
PS	$x = 0$	$x = 1$
A	$C, 1$	$A, 0$
B	$D, 0$	$D, 0$
C	$A, 0$	$D, 0$
D	$B, 0$	$C, 0$

M_1

	NS, z	
PS	$x = 0$	$x = 1$
A	$D, 0$	$C, 1$
B	$A, 0$	$B, 1$
C	$E, 0$	$B, 1$
D	$B, 0$	$D, 1$
E	$C, 1$	$E, 1$

M_2

	NS, z	
PS	$x = 0$	$x = 1$
A	$A, 0$	$E, 1$
B	$E, 1$	$A, 0$
C	$F, 1$	$B, 0$
D	$B, 0$	$F, 1$
E	$C, 1$	$G, 0$
F	$G, 0$	$C, 1$
G	$H, 0$	$D, 1$
H	$D, 1$	$H, 0$

M_3

Problem 13.10

(a) Find a preset distinguishing experiment that determines the initial state of the machine shown in Table P13.10, given that it cannot initially be in state E.

(b) Can you identify the initial state when the initial uncertainty is $(ABCDE)$?

Table P13.10

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 1$	$A, 1$
B	$E, 0$	$A, 1$
C	$A, 0$	$E, 1$
D	$C, 1$	$D, 1$
E	$E, 0$	$D, 1$

Problem 13.11. Specify the entries marked * in the machine of Table P13.11 in such a way that the machine will be strongly connected and the sequences 000 and 111 will be distinguishing sequences.

Table P13.11

	NS, z	
PS	$x = 0$	$x = 1$
A	$*, 0$	$*, 0$
B	$C, 0$	$D, 0$
C	$A, 0$	$B, 0$
D	$D, 1$	$A, 1$

Problem 13.12. Prove that the length L of the minimal distinguishing sequence for a machine with n states and q output symbols is bounded by

$$L \geq \frac{\log_2 n}{\log_2 q}.$$

Problem 13.13. Let M be a reduced n-state machine with input alphabet $I = \{I_1, I_2, \ldots, I_p\}$.

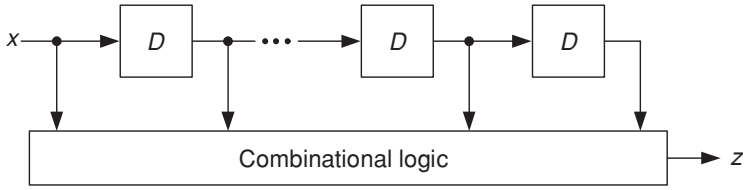
(a) Prove that if, for every input symbol I_i in M, there exists a pair of states whose successors are identical while producing the same output symbol in response to I_i then M does not have any distinguishing sequence.

(b) Prove that if there exists no such pair of states as that described in (a) for any input symbol I_i in M then M has a preset distinguishing sequence whose length is at most $\frac{1}{2}n(n-1)$.

Problem 13.14

(a) (a) Show that every machine of the form in Fig. P13.14 has a synchronizing sequence. Find such a sequence and specify its length.

Fig. P13.14



(b) Does every machine of this form also have a distinguishing sequence? Prove that it does or show a counter-example.
(c) Can every finite-state machine be realized in this form?

Problem 13.15. The response of the machine shown in Table P13.15 to an unknown input sequence is given to an experimenter. Devise a procedure that the experimenter may use in order to identify the initial state. What are the minimum-length sequences that will make such an identification possible?

Table P13.15

	NS, z	
PS	$x = 0$	$x = 1$
A	$A, 0$	$B, 0$
B	$C, 0$	$D, 0$
C	$D, 1$	$C, 1$
D	$B, 1$	$A, 1$

Problem 13.16. The machine shown in Table P13.16 is initially provided with an input sequence 01 to which it responds by producing an output sequence 10. It is next provided with the sequence 1010101010010011010001. Assuming that no fault increases the number of states, show that this sequence is a checking experiment for this machine and find the correct output sequence.

Table P13.16

	NS, z	
PS	$x = 0$	$x = 1$
A	$A, 1$	$B, 0$
B	$C, 0$	$A, 0$
C	$B, 0$	$C, 1$

Problem 13.17. The initial state of the machine shown in Table P13.17 is A, but its entry in row D, column 1, is unknown. An input sequence 0110 was applied to the machine, which produced an output sequence whose last two symbols are 00. Following this sequence, a sequence 101 was applied, and this in turn produced an output sequence whose last symbol is a 0. Determine the missing entry.

Table P13.17

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 0$	$C, 1$
B	$A, 1$	$D, 1$
C	$C, 0$	$A, 1$
D	$E, 1$	$*$
E	$A, 0$	$E, 0$

Problem 13.18. The input sequence X shown below was applied to a reduced five-state machine whose state table is to be determined. In response, the machine produced output sequence Z. Give the state table of the machine in standard form if its starting state is A.

X:	0	0	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0	0	1	0	0	1	0
Z:	0	1	2	0	1	3	2	1	1	0	1	3	3	2	0	1	3	3	3	2	1	2	1	1

Problem 13.19. Construct a checking experiment for the machine of Table P13.19. (Such an experiment need not require more than 24 symbols.)

Table P13.19

	NS, z	
PS	$x = 0$	$x = 1$
A	$D, 0$	$C, 0$
B	$C, 0$	$D, 0$
C	$A, 0$	$B, 0$
D	$D, 1$	$A, 1$

Problem 13.20. The following experiment was proposed as a checking experiment for the machine shown in Table P13.20, when started in state A and under the assumption that the number of states will not increase as a result of a fault. Either prove that it is a proper checking experiment, i.e., that it identifies the machine uniquely, or show by

Table P13.20

	NS, z	
PS	$x = 0$	$x = 1$
A	$A, 2$	$B, 2$
B	$C, 0$	$A, 1$
C	$D, 1$	$E, 0$
D	$E, 2$	$A, 0$
E	$B, 1$	$C, 2$

means of a counter-example that it is not such an experiment.

Input :	0	0	1	0	0	1	0	1	0	1	1	0	0	0	1	0	0
Output :	2	2	2	0	1	0	2	2	0	0	2	1	2	1	1	2	2

Problem 13.21. A four-state machine received the input sequence X shown below and, in response, produced output sequence Z.

(a) What are the distinguishing sequences for the machine?

(b) Assuming the machine starts in state A, do the sequences below correspond to a unique machine? If yes, show its state table; if not, show all possible state tables.

X :	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1
Z :	0	0	1	1	0	0	1	1	1	0	0	1	1	0	0	1	1	0

Problem 13.22. By referring to the machine in Table P13.22, where $\lfloor g \rfloor$ is the largest integer not exceeding g, prove that the bound established in Section 13.6 for definite diagnosability is the least upper bound. That is, prove that for every n there exists an n-state machine, as given in Table P13.22, which is definitely diagnosable and of order $\mu = \frac{1}{2}n(n-1)$.

Table P13.22

PS	I_1	I_2	I_3
1	2, 0	3, 0	2, 0
2	3, 0	4, 0	3, 0
3	4, 0	5, 0	4, 0
⋮	⋮	⋮	⋮
i	$i+1, 0$	$i+2, 0$	$i+1, 0$
⋮	⋮	⋮	⋮
$\lfloor n/2 \rfloor - 1$	$\lfloor n/2 \rfloor, 0$	$\lfloor n/2 \rfloor + 1, 0$	$\lfloor n/2 \rfloor, 0$
$\lfloor n/2 \rfloor$	$\lfloor n/2 \rfloor + 1, 0$	$\lfloor n/2 \rfloor + 2, 1$	$\lfloor n/2 \rfloor + 1, 1$
⋮	⋮	⋮	⋮
j	$j+1, 0$	$j+2, 1$	$j+1, 1$
⋮	⋮	⋮	⋮
$n-2$	$n-1, 0$	$n, 1$	$n-1, 1$
$n-1$	$n, 0$	1, 1	$n, 0$
n	1, 1	1, 0	$n, 1$

Problem 13.23

(a) Show the testing table and graph for the machine given in Table P13.23.

(b) Add to the machine one output terminal such that the sequence 11 becomes a distinguishing sequence.

(c) Design a checking experiment for the augmented machine. (Twenty four symbols are sufficient.)

Table P13.23

	NS, z	
PS	$x = 0$	$x = 1$
A	$A, 0$	$B, 0$
B	$A, 0$	$C, 0$
C	$A, 0$	$D, 0$
D	$A, 1$	$A, 0$

Problem 13.24. An unknown three-state machine with two input symbols 0 and 1 is provided with input sequence X, and it responds by producing output sequence Z. These sequences are given below:

X : 1 1 0 0 1 0 1 0 1 1 1 1 1 0 0 0 1 1 0 0 1 0 1
Z : 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 1 1 0 0 0

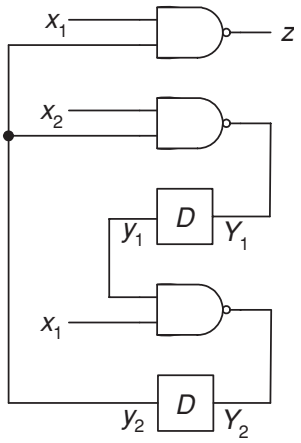
Show that this experiment is sufficient to identify the machine uniquely (up to isomorphism).

Problem 13.25. For the machine M_5 shown in Table 13.10:
(a) obtain a minimum set of collapsed SST faults;
(b) derive a test sequence for the SST fault that corrupts $< 1, B, B, 1 >$ to $< 1, B, C, 1 >$.

Problem 13.26. For the circuit in Fig. 13.7(a):
(a) find a test sequence for the fault y s-a-1 using the extended D-algorithm;
(b) repeat (a) for the fault y s-a-0.

Problem 13.27. Consider the sequential circuit shown in Fig. P13.27. Suppose that it is to be tested for all single stuck-at faults in its combinational logic using full scan.
(a) Find a minimal test set for its combinational logic.
(b) What is the minimum number of clock cycles needed to apply all vectors from your test set to the circuit using scan?

Fig. P13.27



Problem 13.28

(a) How many loops does the state diagram of the LFSR based on feedback polynomial $p(x) = x^4 + x^2 + 1$ consist of?

(b) Find a primitive polynomial of degree 4, and show that the state diagram of the corresponding LFSR consists of only two loops, one with the all-0 state and the other with all the remaining states.

Problem 13.29

(a) Consider an LFSR based on a primitive polynomial. Prove that if its seed is the all-0 state then it remains in the all-0 state.

(b) Show how one can modify the design of such a k-stage LFSR such that it can generate all the 2^k states in one loop in its state diagram.
Hint: The addition of a $(k-1)$-input NOR gate and a two-input EXCLUSIVE-OR gate to the design shown in Fig. 13.14 is enough.

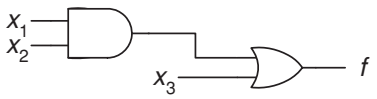
(c) Verify that your modification of the LFSR shown in Fig. 13.15 generates all eight states in a loop.

Problem 13.30. Consider the sequence of test patterns generated by a k-stage LFSR with a feedback polynomial $p(x)$, where the values at $y_k, y_{k-1}, \ldots, y_1$ are said to said to constitute a test pattern. The above sequence of patterns can be generated in reverse order if the k-stage LFSR is based on the feedback polynomial $x^n p(1/x)$ instead and the values at $y_1, y_2, \ldots, y_k$ are said to constitute a test pattern. For example, $x^3 + x^2 + 1$ and $x^3 + x + 1$ form such a pair of polynomials. Verify the above assertion for the LFSRs based on this pair by comparing their state diagrams.

Problem 13.31. Suppose the circuit given in Fig. P13.31 is to be tested by the LFSR shown in Fig. 13.15 for all single stuck-at faults. Derive a stuck-at fault test set for this circuit such that this test set can be applied to it in four clock cycles from the LFSR, starting from a particular seed. Assume that y_1 is connected to x_1, y_2 to x_2, and y_3 to x_3.

Hint: No re-seeding is necessary.

Fig. P13.31



CHAPTER

14 Memory, definiteness, and information losslessness of finite automata

An important characteristic of a finite-state machine is that it has a "memory," i.e., the behavior of the machine is dependent upon its past history. While the behavior of some machines depends on remote history, the behavior of others depends only on more recent events. The amount of past input and output information needed to determine the machine's future behavior is called the *memory span* of the machine.

If the initial state of a deterministic completely specified machine and the input sequence to it are known then the corresponding final state and output sequence can be determined uniquely. However, there are special situations in which either the initial state is unknown or some past input symbols are unknown. In such situations, the behavior of the machines cannot always be predicted in advance. In this chapter, we shall try to answer the following questions. For a given machine, what is the minimum amount of past input–output information required in order to render its future behavior completely predictable? Under what conditions can the input sequence to the machine be reconstructed from its output sequence? Finally, we shall investigate some aspects of the relationship between finite-state machines and coding theory.

14.1 Memory span with respect to input–output sequences (finite-memory machines)

A finite-state machine M is defined as a *finite-memory machine of order* μ, if μ is the least integer such that the present state of M can be determined uniquely from the knowledge of the last μ input symbols and the corresponding μ output symbols. In other words, a machine is finite-memory of order μ if and only if every input sequence of length μ is a homing sequence. Consequently, the homing tree can serve as a possible tool for the detection and recognition of a finite memory for M. In this section, however, we shall derive

Table 14.1 Machine M_1

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 0$	$C, 1$
B	$D, 0$	$C, 0$
C	$D, 0$	$B, 1$
D	$C, 0$	$A, 0$

Table 14.2 Testing table for M_1

PS	0/0	0/1	1/1	1/0
A	B	—	C	—
B	D	—	—	C
C	D	—	B	—
D	C	—	—	A
AB	BD	—	—	—
AC	BD	—	BC	—
AD	BC	—	—	—
BC	DD	—	—	—
BD	CD	—	—	AC
CD	CD	—	—	—

a different test, which will be shown to be valid for all memory aspects of automata.

The testing table and testing graph[1]

Consider a machine M_1, whose state table is shown in Table 14.1. We may rewrite that state table as shown in the upper half of Table 14.2. The column headings of Table 14.2 consist of all input–output symbol combinations, and the entries of the upper half of the table are the next-state entries corresponding to these combinations. For example, the 1-successor of state C is B, and the corresponding output symbol is $z = 1$. Consequently, a B is entered in row C, column $1/1$, of the table, and a dash (—) is entered in row C, column $1/0$. The entire upper half of Table 14.2 is completed in a similar manner.

The row headings in the lower half of the table are all the unordered pairs of states, while the table entries are the corresponding successors. If the entries in rows S_i and S_j, column I_k/O_l, of the upper half are S_p and S_q respectively then the entry in row S_iS_j, column I_k/O_l, of the lower half is S_pS_q. For example, the entries in rows A and C, column $1/1$, are C and B, respectively. Consequently, the entry in row AC, column $1/1$, is BC. If, for some pair of states S_i and S_j, either one or both corresponding entries in some column I_k/O_l are dashes then the entry in row S_iS_j, column I_k/O_l, is a dash. For example, the entry in row AB, column $1/0$, is a dash since the entry in row A, column $1/0$, is a dash, and so on. The table so completed is called a *testing table for finite memory*, or simply, a *testing table*.

We shall refer to a pair of states (S_iS_j) as an *uncertainty pair*, and to its successor (S_pS_q) as the *implied pair*. Thus, for example, the pair (AC) is implied by (BD).

[1] The testing table and graph are similar to those presented in Section 13.6, but are redefined here for completeness of the presentation.

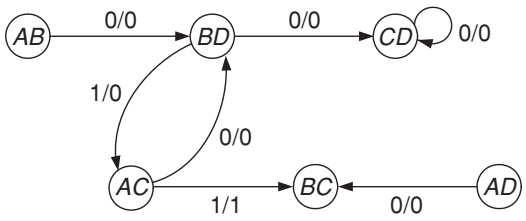


Fig. 14.1 The testing graph G_1 for M_1.

Let us now define a directed graph G, which will be called a *testing graph* (*for finite memory*), in the following way.

1. Corresponding to each row in the lower half of the testing table, there is a vertex in G. The vertex label is the same as the row heading.
2. An arc is drawn leading from the vertex labeled $S_i S_j$ to the vertex labeled $S_p S_q$, where $p \neq q$, if and only if there exists an entry $S_p S_q$ in row $S_i S_j$, column I_k / O_l, of the testing table. The arc is labeled I_k / O_l. No arc is needed if $S_i S_j$ implies $S_p S_p$, e.g., DD in row BC.

The testing graph G_1 for machine M_1 is derived directly from the lower half of the testing table and is shown in Fig. 14.1.

Conditions for finite memory

Let the initial uncertainty regarding the state of machine M be $(S_1 S_2 \cdots S_n)$. M is finite-memory of order μ if the application of any input sequence of length μ transfers the machine into an identifiable state, and if there exists an input sequence of length $\mu - 1$ that, together with the corresponding output sequence, does not provide enough information for a unique identification of the final state.

Theorem 14.1 *A sequential machine M has a finite memory if and only if its testing graph G is loop-free.*

Proof Assume that G is not loop-free. Then, by repeatedly applying the symbols coinciding with the labels of the arcs in the loop, we can find an arbitrarily long input sequence that cannot resolve the uncertainty regarding the final state, thus the machine is not finite-memory. To prove sufficiency, assume that G is loop-free. If M is not finite-memory then there exists an arbitrarily long path in G corresponding to some input sequence X and some pair of states $(S_i S_j)$ such that S_i and S_j cannot be distinguished by X. However, since the number of vertices in G cannot exceed $\frac{1}{2}(n-1)n$ (corresponding to the number of distinct pairs of states), arbitrarily long paths in G are possible only if it contains a loop. Thus, the theorem is proved. ◇

Table 14.3 Machine M_2

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 0$	$D, 0$
B	$C, 0$	$C, 0$
C	$D, 0$	$A, 0$
D	$D, 0$	$A, 1$

Table 14.4 Testing table for M_2

PS	0/0	0/1	1/1	1/0
A	B	—	—	D
B	C	—	—	C
C	D	—	—	A
D	D	—	A	—
AB	BC	—	—	CD
AC	BD	—	—	AD
AD	BD	—	—	—
BC	CD	—	—	AC
BD	CD	—	—	—
CD	DD	—	—	—

Example From the testing graph of M_1 (Fig. 14.1), it is evident that, since G_1 contains two loops, M_1 is not finite-memory. An arbitrarily long string of 0 input symbols will never resolve the uncertainty (CD). Similarly, if the initial uncertainty is (AC) then the input sequence $0101\cdots01$ will transfer the machine to (BD), (AC), (BD), ..., and so on.

Corollary *Let G be a loop-free testing graph for machine M. If the length of the longest path in G is l then $\mu = l + 1$.*

Proof Since G is loop-free, M has a finite memory. Assume that $\mu > l + 1$; then there exists at least one uncertainty pair $(S_i S_j)$ that is transferred, by the application of an input sequence of length $l + 1$, to another pair $(S_p S_q)$. Consequently, there must exist a path between vertices $S_i S_j$ and $S_p S_q$ in G whose length is $l + 1$. This contradicts our assumption and thus μ cannot exceed $l + 1$. The proof that μ cannot be smaller than $l + 1$ is trivial. ◇

From the preceding results, it is evident that *if a machine is finite-memory of order μ then $\mu \leq \frac{1}{2}(n-1)n$.*

A machine for which $\mu = \frac{1}{2}(n-1)n$

The machine M_2 shown in Table 14.3 illustrates the case where the bound of μ is achieved. The corresponding testing table and graph are given in Table 14.4 and Fig. 14.2, respectively.

Clearly, the testing graph of M_2 is loop-free and its maximal path, emanating from AB and terminating at CD, is of length 5. Hence, $\mu = 6$. In general, it can be shown (see Problem 14.3) that there exists a class of machines for which $\mu = \frac{1}{2}(n-1)n$ and, therefore, the bound of μ is the least upper bound and cannot be improved.

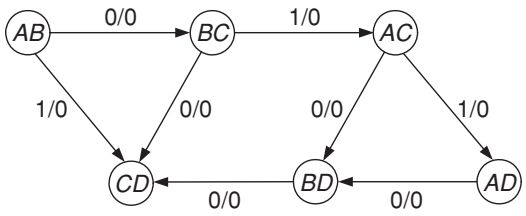


Fig. 14.2 Testing graph for M_2.

*An algorithm to determine whether a graph is loop-free

When the number of vertices in a testing graph G is large, it is desirable to have a more systematic algorithm to determine whether it is loop-free and, if it is, the length of the longest path l. We present here one such algorithm, which does not require the actual drawing of the graph and can be easily executed by a computer.

Let G be a directed graph with p vertices. Define the *connection matrix* of G to be a $p \times p$ matrix whose (i, j)th entry is 1 if there is an arc emanating from vertex i and terminating at vertex j, and is 0 otherwise. The labels associated with the rows and columns of the matrix are the same as the labels of the vertices of G. The labels associated with corresponding rows and columns are identical; i.e., the ith row and the ith column have the same label.

The procedure for determining whether a graph is loop-free can be illustrated by means of the machine M_2. The connection matrix of M_2 is derived directly from the testing table and is as follows:

$$\begin{array}{l} (AB) \\ (AC) \\ (AD) \\ (BC) \\ (BD) \\ (CD) \end{array} \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Two arcs emanate from vertex AB: to BC and CD. Therefore, the entries in row AB, columns BC and CD, are 1, and so on.

If a directed graph G is loop-free then it has one or more terminal vertices.[2] Furthermore, the subgraph resulting from the removal of a terminal vertex and all arcs leading to it is also loop-free. This can be proved by observing that if G has no terminal vertex then we can construct arbitrarily long paths in G. However, since G is finite, this means that G has a loop. In the matrix representation, the removal of a vertex and all arcs leading to and from it is accomplished by the deletion from the matrix of the row and column corresponding to this vertex.

[2] A vertex from which no arcs emanate is called a *terminal vertex*.

The testing algorithm is summarized as follows.

1. Given a testing table, construct the corresponding connection matrix.
2. Delete all the rows having 0's in all positions and remove the corresponding columns. If there are none, go to step 4.
3. Repeat step 2.
4. If the matrix has not completely vanished then G is not loop-free. If the matrix has vanished, G is loop-free. (A "vanished" matrix has no rows or columns.)

Returning to the connection matrix of M_2, the first application of step 2 results in the removal of the row labeled (CD) and its corresponding column. The resulting matrix is

$$\begin{matrix}(AB)\\(AC)\\(AD)\\(BC)\\(BD)\end{matrix}\begin{bmatrix}0&0&0&1&0\\0&0&1&0&1\\0&0&0&0&1\\0&1&0&0&0\\0&0&0&0&0\end{bmatrix}.$$

Repeated applications of step 2 result in the removal of the rows labeled (BD), (AD), (AC), and so on:

$$\begin{matrix}(AB)\\(AC)\\(AD)\\(BC)\end{matrix}\begin{bmatrix}0&0&0&1\\0&0&1&0\\0&0&0&0\\0&1&0&0\end{bmatrix}\begin{matrix}(AB)\\(AC)\\(BC)\end{matrix}\begin{bmatrix}0&0&1\\0&0&0\\0&1&0\end{bmatrix}\begin{matrix}(AB)\\(BC)\end{matrix}\begin{bmatrix}0&1\\0&0\end{bmatrix}(AB)\begin{bmatrix}0\end{bmatrix}.$$

Clearly, at the next step the matrix vanishes.

We observe that at each application of step 2 we remove the terminal vertices and all arcs leading to them. Consider the terminal vertices at the end of the longest paths whose length is l. It takes $l+1$ applications of step 2 to remove all the vertices in these paths and to eliminate the matrix. Consequently, *the number of times that step 2 is applied is equal to order μ of the memory*. In the preceding example, step 2 was applied six times; consequently, M_2 is finite-memory of order $\mu = 6$, as is already known. Note that if at some time the matrix contains two (or more) rows consisting of 0's in all their positions, all these rows and their corresponding columns must be deleted simultaneously, and this step counts as a single application of step 2.

14.2 Memory span with respect to input sequences (definite machines)

A sequential machine M is called a *definite machine of order μ* if μ is the least integer such that the present state of M can be determined uniquely from knowledge of the last μ input symbols to M. A definite machine is thus said to

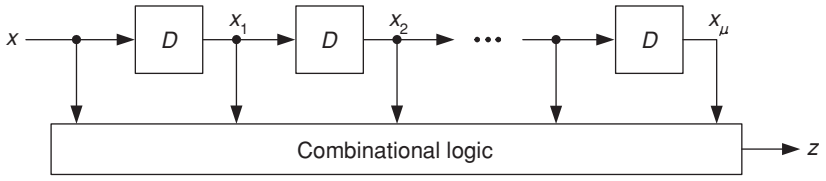


Fig. 14.3 Canonical realization of a μ-definite machine.

have a finite input memory. However, for a nondefinite machine there always exists at least one input sequence of arbitrary length that does not provide enough information to identify the state of the machine. A definite machine of order μ is often called a *μ-definite machine*. Clearly, if a machine is μ-definite then it is also finite-memory of order equal to or smaller than μ.

The knowledge of any μ past input values is always sufficient to completely specify the present state of a μ-definite machine. Therefore, any μ-definite machine can be realized as a cascade connection of μ delay elements, which store the last μ input values, and a combinational circuit that generates the specified output value. This realization, which is often referred to as the *canonical realization of a definite machine*, is shown in Fig. 14.3.

Properties of definite machines

We shall now study some properties of definite machines, from which we shall derive tests for definiteness. The first obvious property is that a machine is definite of order μ if and only if every sequence of length μ is a synchronizing sequence. This property can be detected by means of the synchronizing tree presented in Section 13.2. The tree is terminated whenever either of the following occurs.

1. An uncertainty in the kth level is also associated with some node in a preceding level.
2. All nodes of the kth level are associated with singleton uncertainties, i.e., uncertainties that consist of a single state each.

Clearly, if the tree terminates by virtue of rule 1 then the corresponding machine is not definite. However, if the tree terminates by virtue of rule 2 then the corresponding machine is definite, since this means that every input sequence (i.e., path in the tree) leads to a unique final state. Furthermore, the length of the path determines the order of definiteness; that is, if the tree is terminated in level k and rule 2 is satisfied then the corresponding machine is k-definite. Note that if some node is associated with a singleton uncertainty then that node may become terminal, but the successors of other nodes must be determined. The order of definiteness is determined by the length of the longest path.

Example Consider the machine M_3 whose state table is given in Table 14.5. The output entries have been omitted, since only the inputs to the machine play a role in the determination of definiteness. The synchronizing tree for machine M_3 is shown in Fig. 14.4. Its length is $k = 3$ and, consequently, M_3 is definite of order 3.

Table 14.5 Machine M_3

	NS	
PS	$x = 0$	$x = 1$
A	A	B
B	C	B
C	A	D
D	C	B

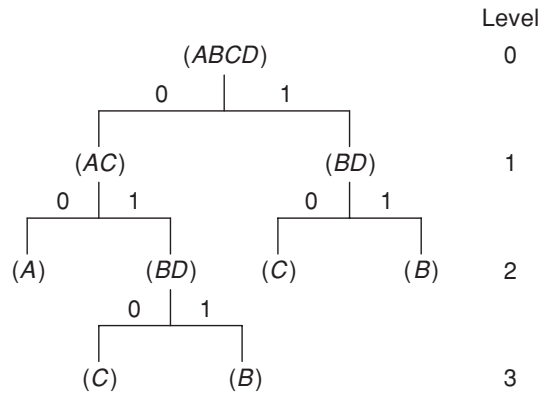


Fig. 14.4 Synchronizing tree for M_3.

Let M be a μ-definite machine, and let $(S_i S_j)$ be a nontrivial uncertainty in the $(\mu - 1)$th level of the corresponding synchronizing tree. Since the μth level of the tree consists of only single states, the I_k-successors of both S_i and S_j must be identical for every possible I_k in I; that is, every definite machine contains at least two distinct states for which $I_k S_i = I_k S_j$ for all I_k in I. Define the *contracted table* $\overline{M}$ as the table obtained by deleting row S_j and replacing in the entire table all appearances of S_j by S_i. It is easy to show that the application of any input sequence X to $\overline{M}$ or M, when initially in any state S_k such that $S_k \neq S_j$, will pass both $\overline{M}$ and M to the same final state if the final state is different from S_j and will pass $\overline{M}$ to S_i if the final state of M is S_j.

More generally, let $\overline{M}$ be the contracted table obtained from M by replacing each set of states whose I_k-successors are identical by a single member from that set. Clearly, the synchronizing tree of $\overline{M}$ has only $\mu - 1$ levels, and its

last level consists of only singleton uncertainties. However, since such a tree corresponds to a machine which is $(\mu - 1)$-definite, we arrive at the following general result.

- If M is a μ-definite machine then the contracted machine $\overline{M}$ is $(\mu - 1)$-definite. Conversely, if $\overline{M}$ is k-definite then M is $(k + 1)$-definite. If $\overline{M}$ is not definite, neither is M.

Tests for definiteness

The synchronizing tree can be used to test for definiteness. In this section we shall illustrate two additional testing procedures. The first procedure, which utilizes the previously derived properties of definite machines, involves repeated derivations of contracted tables. The second procedure is based on the familiar testing graph.

The first test for the definiteness of a machine M is as follows.

1. Determine the subsets of states whose I_k-successors are identical.
2. Select one representative state in each subset.
3. Obtain the contracted table $\overline{M}$ by replacing each subset with its representative and modifying the table entries accordingly.
4. Regard $\overline{M}$ as a new table and repeat the previous steps until no new contractions are possible.

The machine M is definite if and only if the final contracted table obtained in step 4 consists of just a single state.

Example The machine M_4 of Table 14.6 will be tested for definiteness. The nontrivial subsets of states whose corresponding successors are identical are (B, F) and (C, D). Select B and C as the representative states and obtain the contracted table $\overline{M}_4$, which consists of four states as shown in Table 14.7. States B and C in the contracted table can now be represented by

Table 14.6 Machine M_4

	NS	
PS	$x = 0$	$x = 1$
A	A	B
B	E	B
C	E	F
D	E	F
E	A	D
F	E	B

Table 14.7 The contracted machine $\overline{M}_4$

	NS	
PS	$x = 0$	$x = 1$
A	A	B
B	E	B
C	E	B
E	A	C

Table 14.8 Repeated contractions of M_4

	NS	
PS	$x=0$	$x=1$
A	A	B
B	E	B
E	A	B

(a)

	NS	
PS	$x=0$	$x=1$
A	A	B
B	A	B

(b)

	NS	
PS	$x=0$	$x=1$
A	A	A

(c)

state B, and the contracted table shown in Table 14.8a results. The fourth contraction yields a single-state machine. Thus, M_4 is definite.

We shall now show that the test for definiteness is always finite, and determine the bound on its length.

Theorem 14.2 *Given that a machine M is μ-definite, $\mu \leq n-1$, where n is the number of states of the machine. Moreover, the order of definiteness is equal to the number of contractions needed to obtain a one-state machine.*

Proof Since M is μ-definite, $\overline{M}$ is $(\mu-1)$-definite. Each contracted table must contain at least one state less than its predecessor. Consequently, after at most $n-1$ repeated contractions we obtain a one-state machine that is 0-definite, i.e., no input symbol is required in order to determine its present or final state. To determine the order of definiteness, it is necessary to count backward; that is, the last contracted table is 0-definite, its predecessor is 1-definite, and so on. ◇

For machine M_4, $\mu = 4$ since four contractions are necessary in order to obtain a one-state machine.

The second test for definiteness is based on a testing table and graph, which are defined as follows. The *testing table* (*for definiteness*), which is divided into two parts, has p columns corresponding to $I_1, I_2, \ldots, I_p$. The rows in the upper part of the table correspond to the states of the machine, and the table entries are the state transitions. The row headings in the lower part of the table are all unordered pairs of states, while the table entries are the corresponding successors. The *testing graph* (*for definiteness*) is defined as in the previous section and is derived directly from the lower part of the testing table. The arc labels, however, are now input symbols instead of input–output symbol combinations.

Example The testing table for the machine M_3 is shown in Table 14.9, and the corresponding testing graph, which is loop-free, is shown in Fig. 14.5.

Table 14.9 The testing table for M_3 (see Table 14.5)

PS	$x=0$	$x=1$
A	A	B
B	C	B
C	A	D
D	C	B
AB	AC	BB
AC	AA	BD
AD	AC	BB
BC	AC	BD
BD	CC	BB
CD	AC	BD

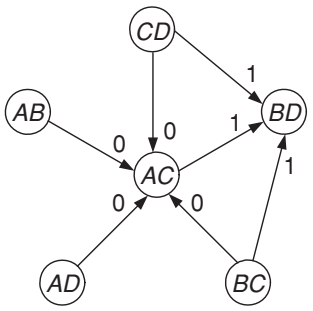


Fig. 14.5 Testing graph for M_3.

Theorem 14.3 *A machine M is μ-definite if and only if its corresponding testing graph G is loop-free. If the length of the longest path in G is l then $\mu = l + 1$.*

Proof The proof is similar to that of Theorem 14.1 and is left to the reader as an exercise. ◇

The machine M_3 is definite of order $\mu = 3$, since its testing graph is loop-free and the longest path in the graph is of length $l = 2$.

The relationship between the testing graph and the synchronizing tree is evident. A loop-free graph means that no uncertainty in the kth level of the tree is also associated with some node in a preceding level and, conversely, a loop in the graph means that such a situation does occur.

14.3 Memory span with respect to output sequences

A finite-state machine M is said to have an *output memory of order* μ if μ is the least integer such that the knowledge of the last μ output symbols suffices to determine the state of M at some time during the last μ transitions. In this section, emphasis is placed on the specification of the state of M at *some time* during the experiment, instead of on the identification of the final state. The case of identifying the final state is more restricted and is left to the reader as an exercise.

Test for output memory

The major tools for testing whether a given machine has a finite output memory are a modified testing table and its corresponding testing graph. The *testing table* (*for output memory*), which consists of two parts, has q columns corresponding to the output symbols of the machine, i.e., $O_1, O_2, \ldots, O_q$. The row names of

Table 14.10 Machine M_5

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 0$	$D, 1$
B	$C, 1$	$A, 1$
C	$B, 0$	$C, 0$
D	$C, 0$	$C, 1$

Table 14.11 Testing table for M_5

PS	$z = 0$	$z = 1$
A	B	D
B	—	(AC)
C	(BC)	—
D	C	C
AB	—	$(AD)(CD)$
AC	$(BB)(BC)$	—
AD	(BC)	(CD)
BC	—	—
BD	—	$(AC)(CC)$
CD	$(BC)(CC)$	—

the upper part of the table are the states of M. The entries in row S_i, column O_j, are the states that can be reached from S_i by single transitions associated with the output symbol O_j. We shall call these states the (*output*) O_j*-successors* of S_i. The entire upper half of the testing table is, actually, a listing of the output successors of the states of M and is therefore called an *output successor table*. Thus, for the machine M_5 of Table 14.10, the output 1-successors of B are A and C; state B has no output 0-successors. This is recorded in Table 14.11 by entering AC in row B, column 1, and a dash in row B, column 0. When the reference to output successors is self-evident in the context, we shall omit the adjective "output."

For each unordered pair of states there is a row in the lower half of the testing table. The table entries are the corresponding output successors. The output O_k-successors of S_iS_j are all pairwise combinations of the output O_k-successors of S_i and S_j. For example, if the successors of S_i and S_j are S_pS_q and S_rS_t respectively then the corresponding successors of S_iS_j are S_pS_r, S_pS_t, S_qS_r, S_qS_t. If, for some pair of states S_i and S_j, either one or both O_k-successors are dashes then the O_k-successor of S_iS_j is also a dash. Thus, since the output 1-successor of C is a dash, the output 1-successor of AC is also a dash, as shown in the lower half of Table 14.11.

A *testing graph* (*for output memory*) G is a directed graph, such that:

1. corresponding to each row in the lower half of the testing table there is a vertex in G, whose label is the same as the row heading;
2. an arc labeled O_k is drawn from vertex S_iS_j to vertex S_pS_q, where $p \neq q$, if and only if S_pS_q is an entry at row S_iS_j, column O_k.

The testing graph of the machine M_5 is shown in Fig. 14.6. Note that two or more arcs having the same label may emanate from a single vertex, e.g., vertex AB.

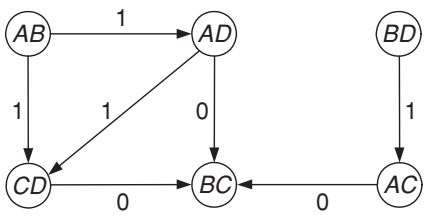


Fig. 14.6 Testing graph G_5 for M_5.

Theorem 14.4 *A finite-state machine M has a finite output memory if and only if its corresponding testing graph G is loop-free. Furthermore, if G is loop-free and the longest path in G is of length l then M has an output memory of order* $\mu = l + 1$.

Proof If G contains a loop, choose any two vertices in the loop, say $S_i S_j$ and $S_p S_q$; then there exist two identical output sequences, produced by M while in transition from S_i via S_p to S_i and from S_j via S_q to S_j. Since these sequences may be repeated as many times as we wish, they will never distinguish the states associated with any vertex contained in the loop and, consequently, M does not have a finite output memory. If G is loop-free but M does not have a finite output memory then, for every possible positive integer μ, there exists a path, emanating from some vertex $S_i S_j$, that does not pass M into an identifiable state. This implies arbitrarily long paths in G. However, since G is finite and loop-free, this cannot be achieved and thus M has a finite output memory.

The proof that $\mu = l + 1$ follows from the same line of argument used in the corollary in Section 14.1. ◇

For example, G_5 in Fig. 14.6 is loop-free and its longest path is of length 3; this is the path from AB through AD and CD to BC. Thus, M_5 has a finite output memory of order $\mu = 4$.

Note that the testing graph does not contain any vertex corresponding to pairs consisting of repeated entries, e.g., BB, etc. The existence of such a pair means, in effect, that there is no uncertainty regarding the state of the machine. Therefore, the deletion of such pairs from the graph (or even from the testing table) does not affect the test for finite output memory.

Determining the state of the machine

If a machine M has a finite output memory, it is possible to determine the state of M at some point during any experiment of length μ. We shall now show how to identify this state when the only available information is the output sequence.

Suppose, for example, that the output sequence produced by the machine M_5, in response to some unknown input sequence, is 1110. Initially, the machine could have been in either state A, B, or D, since no 1 output symbol can be

generated by a transition from state C. Thus, the initial uncertainty is (ABD). From the output successor table, we find that the output 1-successor of A is D, of B is (AC), and of D is C. Consequently, the 1-successor uncertainty of (ABD) is (ACD). (In general, the output successor of a set of states Q is the set consisting of all output successors of the members of Q.) In a similar manner, we find that the 1-successor of (ACD) is (CD), and so on. The next state is clearly C, as shown below:

	A	A	C	C	B
Possible uncertainties	B	C	D		C
	D	D			
Output sequence		1	1	1	0

Note that although the state of M_5 has been identified at one point during the above experiment, the uncertainty increases to (BC) one time unit later.

The reason for suggesting the above definition of output memory, which is somewhat different from those of input–output memory or definiteness, is that the output successor table might have multivalued entries. Therefore, the identification of the state of the machine at some point during the experiment does not guarantee the identification of its successor. All we can say is that, within μ transitions corresponding to any output sequence, there must be at least one time period during which the machine is unambiguously in a certain state, regardless of the initial state.

14.4 Information-lossless machines

A central problem in coding and information transmission is the determination of conditions under which it is possible to reconstruct the input sequence to the machine from the corresponding output sequence. It will be shown that whenever a machine is used as an encoding device (i.e., the machine is provided with an input sequence and its output sequence is the coded message) and when its initial and final states are known, its information losslessness guarantees that the coded message can always be deciphered. Thus, we define a machine M to be (*information*) *lossless* if the knowledge of the initial state, output sequence, and final state is sufficient to determine uniquely the input sequence.

Conditions for lossiness

A machine that is not lossless is said to be *lossy*. A simple example of a lossy machine is one in which, for some state S_i and two distinct input symbols I_p

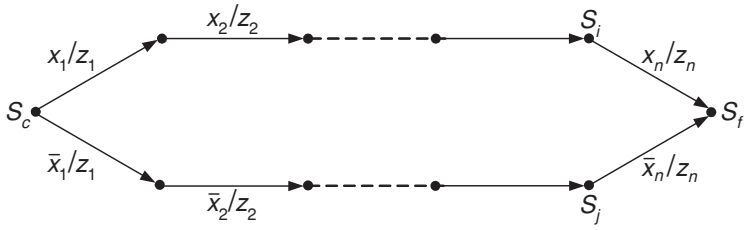


Fig. 14.7 Condition for information loss.

and I_q, the I_p- and I_q-successors and the corresponding output symbols are identical. Clearly, in such a case, knowledge of the output sequence and the initial and final states is not sufficient to determine whether I_p or I_q was applied to the machine.

Loss of information occurs whenever two states, S_i and S_j, which can be reached from a common state S_c by means of two distinct input sequences while producing identical output sequences, merge into a final state S_f and produce the same output sequence. Clearly, once the machine has reached state S_f, no future experiment will make possible the retrieval of the input sequence that transferred M from S_c to S_f. This case, which is necessary and sufficient for a machine to be lossy, is illustrated in Fig. 14.7.

Example The machine M_6 of Table 14.12 is lossy, as demonstrated in Fig. 14.8. Two distinct input sequences (01 and 10) take the machine from state A to state B, while producing identical output sequences (00). After M_6 has reached state B, it is impossible to determine which input sequence actually occurred.

Table 14.12 Machine M_6

	NS, z	
PS	$x = 0$	$x = 1$
A	$A, 0$	$B, 0$
B	$B, 0$	$A, 1$

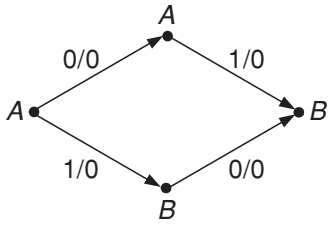


Fig. 14.8 Demonstration that M_6 is lossy.

From the foregoing discussion it is evident that in order to test a machine for losslessness, it is first necessary to determine whether, for a given state, two or more successors and their corresponding output sequences are identical or whether a merger of the type illustrated in Fig. 14.7 exists. Before presenting a test for information losslessness, we shall define an "order" of losslessness.

Table 14.13 Machine M_7

	NS, z	
PS	$x = 0$	$x = 1$
A	$C, 1$	$D, 0$
B	$D, 0$	$A, 1$
C	$D, 1$	$B, 0$
D	$C, 0$	$B, 1$

Information losslessness of finite order

Suppose that a system of lossless machines is used for encoding and decoding purposes. The "encoder" receives an input sequence and, in turn, produces an output sequence, which is transmitted to a "decoder." Clearly, if the encoder is lossless then its input sequence can be reconstructed from its output sequence as well as the information regarding its initial and final states. The major drawback in such a decoding process lies in the fact that the information regarding the final state is transmitted by the encoder only after the entire message has been transmitted. Consequently, the entire message must be stored before the deciphering process can begin. In addition, since the output sequence may be arbitrarily long, the lossless machine cannot serve as a practical tool for encoding and decoding purposes. In view of this limitation, it becomes desirable to look for machines for which it is not necessary to store the entire message, but where the deciphering process can start when only the initial state and a finite length of the output sequence are available.

A machine is said to be (*information*) *lossless of finite order* if the knowledge of the initial state and the first μ output symbols is sufficient to determine uniquely the first input symbol. Knowledge of the initial state and the first input symbol is sufficient to determine the next state, and thus the second input symbol can be computed from the $(\mu + 1)$th output symbol, and so on. The integer μ that is a measure of the delay in the deciphering of the input symbols is said to be the *order* of losslessness if μ is the least integer satisfying the above definition, that is, if for some initial state and a sequence of $\mu - 1$ output symbols there exist at least two possible input sequences that differ in their initial input symbols.

The simplest example of lossless machines of finite order is that of first order, where the first input symbol can be determined from knowledge of the initial state and the first output symbol. Hence, there is no delay in deciphering the input symbols for this class of machines. As an example, consider the machine M_7 shown in Table 14.13. Since for every state of M_7, the output symbol associated with the 0-successor is different from the output symbol associated with the 1-successor, knowing the initial state and first output symbol is sufficient to identify the first input symbol. For example, if M_7 is initially

Table 14.14 Machine M_8

	NS, z	
PS	$x = 0$	$x = 1$
A	$A, 1$	$C, 1$
B	$E, 0$	$B, 1$
C	$D, 0$	$A, 0$
D	$C, 0$	$B, 0$
E	$B, 1$	$A, 0$

Table 14.15 Testing table for M_8

PS	$z = 0$	$z = 1$
A	—	(AC)
B	E	B
C	(AD)	—
D	(BC)	—
E	A	B
AC	—	—
AD	—	—
BC	$(AE)(DE)$	—
AE	—	$(AB)(BC)$
DE	$(AB)(AC)$	—
AB	—	$(AB)(BC)$

in state A and if, in response to an as yet unknown input symbol, output symbol 1 is produced then we can unambiguously identify the input symbol as a 0.

Test for information losslessness

We now derive a test to determine whether a given machine is lossless and to find its order of losslessness if it is finite. Before proceeding with the testing procedure, we introduce some terminology that facilitates discussion on information losslessness. Two states S_i and S_j are said to be (*output*) *compatible* if there exists some state S_p such that both S_i and S_j are its O_k-successors, or if there exists a compatible pair of states S_r, S_t such that S_i, S_j are their O_k-successors. In such a case, we say that the compatible pair $(S_i S_j)$ is *implied* by $(S_r S_t)$.

The first step in the testing procedure is to check each row of the state table for the appearance of two identical next-state entries associated with the same output symbol. If no identical entries appear, the next step is to construct the output successor table. A *testing table* (*for information losslessness*) is now constructed in two parts. The upper part consists of the output successor table, while the lower part is constructed in the following manner. Every compatible pair appearing in the successor table is made a row heading in the lower part of the testing table. The successors of these pairs are found in the usual way; they consist of all implied compatible pairs. Any implied pair that has not yet been used as a row heading is now made a row heading, its successors found, and so on. The process terminates when all compatible pairs have been used as row headings.

The machine M_8 given in Table 14.14 may be used to illustrate the testing procedure. The output successor table is shown in the upper half of Table 14.15. The pair (AC) is compatible, since both A and C are the output 1-successors

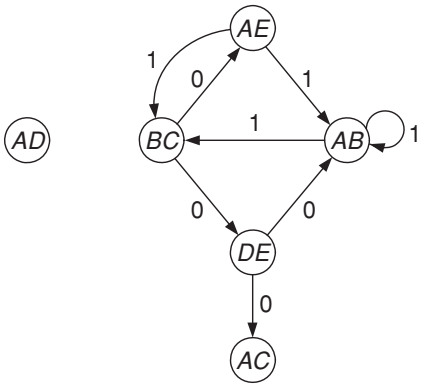


Fig. 14.9 Testing graph G_8 for M_8.

of A. Similarly, the pairs (AD) and (BC) are compatible. Consequently, these pairs are used as row headings for the lower part of the testing table. The pairs (AE) and (DE), which are implied by (BC), are now made row headings, and so on.

Note that, contrary to the testing procedure for finite output memory, the testing table for information losslessness does not necessarily include all distinct pairs of states; it includes only the compatible pairs.

At this point, we are ready to derive necessary and sufficient conditions for a machine to be information lossless. Suppose that the testing table contains a compatible pair consisting of repeated entries, e.g., $(S_k S_k)$; then there exists either some compatible pair $(S_i S_j)$ that implies $(S_k S_k)$ or some state S_i that has identical output successors for two or more input symbols. However, since these cases have been shown to be necessary and sufficient for lossiness, the machine in question must be lossy. We thus arrive at the following general result.

- A machine is lossless if and only if its testing table does not contain any compatible pair consisting of repeated entries.

A *testing graph* (*for information losslessness*) G is a directed graph such that:

1. corresponding to every compatible pair there is a vertex in G;
2. an arc labeled O_k is drawn from vertex $S_i S_j$ to vertex $S_p S_q$, where $p \neq q$, if and only if $(S_p S_q)$ is a compatible implied by $(S_i S_j)$.

The testing graph G_8 of M_8 is derived in the usual way from the lower half of the testing table and is shown in Fig. 14.9. The machine M_8 is clearly lossless, because there are no compatible pairs consisting of repeated entries. Before determining the order of losslessness, we prove the following theorem.

Theorem 14.5 *A machine M is lossless of order $\mu = l + 2$ if and only if its testing graph is loop-free and the length of the longest path in the graph is l.*

Proof Assume that M is lossless. Suppose that G is not loop-free, and let $S_i S_j$ be some vertex in the loop. Clearly, every compatible pair is accessible from some state of M by a pair of distinct input sequences that yield identical output sequences. Thus, we can find a pair of different input sequences that take M to $S_i S_j$ while producing identical output sequences. If we now observe the output symbols that the machine produces while going through all the compatible pairs in the loop, we find that the machine is back in $S_i S_j$ without supplying any additional information to make possible the identification of the first input symbol. In addition, since this loop may be repeated as many times as we wish, we may construct a pair of arbitrarily long input sequences that start in the same state of M and differ in the first symbol but produce identical output sequences. Thus, M is not lossless of finite order. The proof that the loop-free condition is indeed sufficient for finite order is trivial and follows the line of arguments used in the proof of Theorem 14.1.

To determine the order of losslessness, consider the longest path in G. It takes one input symbol to get from a state of M into the first compatible (pair), and it takes l input symbols to go through the longest path in G. Since the compatible that has been reached after $l + 1$ input symbols does not imply any other compatible, one more input symbol will yield different output symbols, depending on which state of the compatible the machine is in. This, in turn, determines the initial input symbol. Thus, $\mu = l + 2$ output symbols (plus the knowledge of the initial state) are sufficient to determine the first input symbol. ◇

From Theorem 14.5 we conclude that if M is lossless of order μ then $\mu \leq 1 + \frac{1}{2}n(n-1)$. The proof that this is indeed the least upper bound is given in Appendix 14.1.

The case $\mu = 1$ is detected by the absence of compatible pairs (see the machine M_7), while the case $\mu = 2$ is detected by the absence of arcs in the graph.

Returning to the machine M_8, we observe that, since G_8 is not loop-free, M_8 is not lossless of finite order. It is interesting to note that M_8 is lossless even though state A can be reached by input symbol 1 from both states C and E and the output symbol produced is 0. This situation does not imply lossiness, since the pair $(C\,E)$ is not compatible, i.e., C and E cannot be reached from any initial state by means of two distinct input sequences while producing identical output sequences.

Example As another illustration, the above test is applied to the machine M_9 of Table 14.16. This machine is shown to be lossless of order 3,

since its testing graph (Fig. 14.10) is loop-free and the longest path is of length 1.

Table 14.16 Machine M_9

	NS, z	
PS	$x = 0$	$x = 1$
A	$A, 0$	$B, 0$
B	$C, 0$	$D, 0$
C	$D, 1$	$C, 1$
D	$B, 1$	$A, 1$

Table 14.17 Testing table for M_9

PS	$z = 0$	$z = 1$
A	(AB)	—
B	(CD)	—
C	—	(CD)
D	—	(AB)
AB	$(AC)(AD)$ $(BC)(BD)$	—
CD	—	$(AC)(AD)$ $(BC)(BD)$

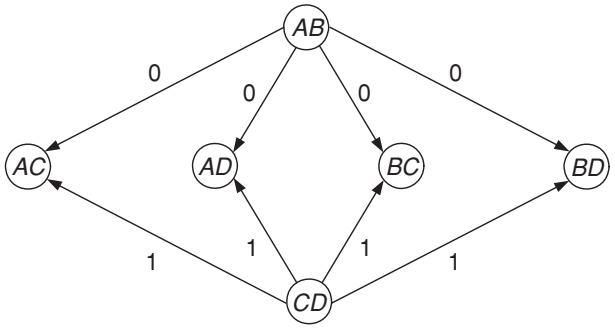


Fig. 14.10 Testing graph for machine M_9.

Retrieval of the input sequence

Knowledge of the output sequence produced by a lossless machine, as well as its initial and final states, is sufficient to determine the input sequence applied to the machine. We shall now present a procedure to retrieve the input sequence by first reconstructing the state sequence. Since the machine is lossless, the input sequence is uniquely specified by the state sequence.

Let M be a lossless machine that is initially in a known state and, after producing a given output sequence of length r, terminates in a known final state. Suppose that we now wish to determine the state of the machine just after it has produced the jth output symbol. By applying the first j output symbols to the output successor table, starting from the known initial state, we can find a set of states in which the machine could be. In an analogous way, we can trace the predecessors of the final state by applying (in reverse order) the $r - j$ output symbols to the output predecessor table (which will be defined shortly). This last step yields a set of possible predecessors just prior to the production of

the $(j+1)$th output symbol. Clearly, since the machine is lossless, there is only one state in which it could have been at the time in question; the intersection of the set derived from the successor table and the set derived from the predecessor table will reveal this state.

As an example, consider the machine M_8 (see Table 14.14). Assume that this machine was initially in state A, has in response to a yet unknown input sequence produced the output sequence 110001100101, and has terminated in state B. From the output successor table (Table 14.15), we find that the 1-successors of A are A and C and the 1-successors of AC are also A and C. Just after the third output symbol, the machine could have been in either state A or D, since AD is the 0-successor of AC. Similar reasoning is used to find the states in which the machine could be after the production of every output symbol. These steps can be summarized as follows, moving from left to right:

$$\begin{array}{cccccccccccccc} & & & & & A & A & A & A & A & A & A & \\ A & A & A & A & B & D & B & B & D & B & B & D & B \\ & C & C & D & C & E & C & C & E & C & C & E & \\ & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{array}$$

We have not yet utilized the information that can be obtained from the final state. This is best accomplished by an (*output*) *predecessor table*, which is constructed as follows. There is a column labeled O_k in the table for each output symbol O_k in O and a row for each state of the machine. The entries in row S_i, column O_k, are those states for which S_i is an output O_k-successor. These states are often referred to as the (*output*) O_k-*predecessors* of state S_i. The output predecessors of each machine state can be found directly from the state table. For convenience, the row headings of the predecessor table are placed on the right-hand side of the table. This emphasizes the fact that the row headings are the successors of the corresponding table entries.

For example, state B of the machine M_8 can be reached by a single transition from states B and E while producing output symbol 1 and from state D while producing output symbol 0. Thus, the entry in row B, column 1, of the output predecessor table (Table 14.18) is BE while the entry in row B, column 0, is D. In a similar manner, we can obtain the entire predecessor table.

Table 14.18 Output predecessor table for machine M_8

$z=0$	$z=1$	NS
CE	A	A
D	BE	B
D	A	C
C	—	D
B	—	E

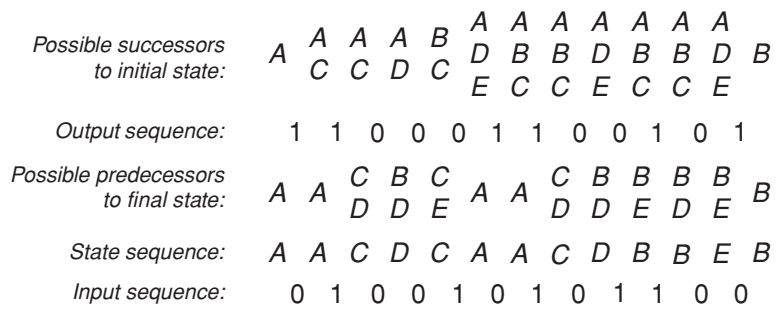


Fig. 14.11 Retrieval of an input sequence.

If we now wish to determine the state of M_8 just prior to the production of the last output symbol, we look for the output 1-predecessors of state B, which is known to be the final state. As shown before, the 1-predecessors of B are B and E. However, from the output successor table we have found that, at the time in question, the machine could have been in one of states A, D, or E. In addition, since it could have been in only one state at that time, this state must be given by the intersection of (B, E) and (A, D, E). Therefore, the 1-predecessor of B is E. The entire procedure is summarized in Fig. 14.11. It is easy to verify by means of the state table that the input sequence that corresponds to the state sequence in Fig. 14.11 is 010010101100.

Whenever a given output sequence has been generated by a lossless machine, the state transitions and input sequence can be determined uniquely. If, however, at some point the intersection of the sets containing the possible successors and predecessors consists of two or more states then there exist at least two distinct input sequences that produce identical output sequences. Therefore, the machine in question is not lossless. If at some point the intersection is empty then the corresponding output sequence could not have been produced by the given machine subject to the specified initial and final states. In fact, if the intersection is empty at one point then it must be empty at all points.

Inverse machines

An *inverse* M^i is a machine which, when excited by the output sequence of a machine M, produces as its output the input sequence to M, after at most a finite delay. Evidently, a deterministic inverse can be constructed only if M is lossless, and it can be constructed such that it produces M's input sequence after just a finite delay if and only if M is lossless of finite order.

Consider, for example, the machine M_7 of Table 14.13, which is lossless of first order. For any possible initial state and output sequence, knowledge of the initial state of M_7 and the first output symbol is sufficient to determine uniquely the first input symbol to the machine. Hence, there is no delay in deciphering the input symbols to this machine. The state transitions of the inverse machine

Table 14.19 Machine M_7^i

	NS, x	
PS	$z = 0$	$z = 1$
A	$D, 1$	$C, 0$
B	$D, 0$	$A, 1$
C	$B, 1$	$D, 0$
D	$C, 0$	$B, 1$

M_7^i are, therefore, given by the output successor table, as shown in Table 14.19. The output symbols associated with these state transitions are found by means of the state table of the machine M_7. If M_7^i is placed in cascade with M_7, it will produce as its output sequence an exact replica of the input sequence to M_7.

For every lossless machine of order μ, knowledge of the state at time $t - \mu + 1$ and of the last μ output symbols, i.e., $z(t - \mu + 1), z(t - \mu + 2), \ldots, z(t)$, is sufficient to determine uniquely the input symbol $x(t - \mu + 1)$. Consequently, if we send the output sequence produced by a lossless machine M of order μ into a register that consists of $\mu - 1$ delay units, we can design a combinational circuit that has as inputs the contents of that register and the state of M at time $t - \mu + 1$ and, in turn, produces the value of $x(t - \mu + 1)$.

The combinational circuit can be specified by a truth table in which the value of $x(t - \mu + 1)$ is specified for every possible combination of $S(t - \mu + 1)$ and $z(t - \mu + 1), z(t - \mu + 2), \ldots, z(t)$. The information regarding the state of M can be supplied to the combinational circuit by a copy of M that is set to be at $t = \mu - 1$ in the same state that M was in at $t = 0$ and receives as its inputs a version delayed by $\mu - 1$ time units of the inputs to M. The schematic diagram of such a deciphering system is shown in Fig. 14.12.

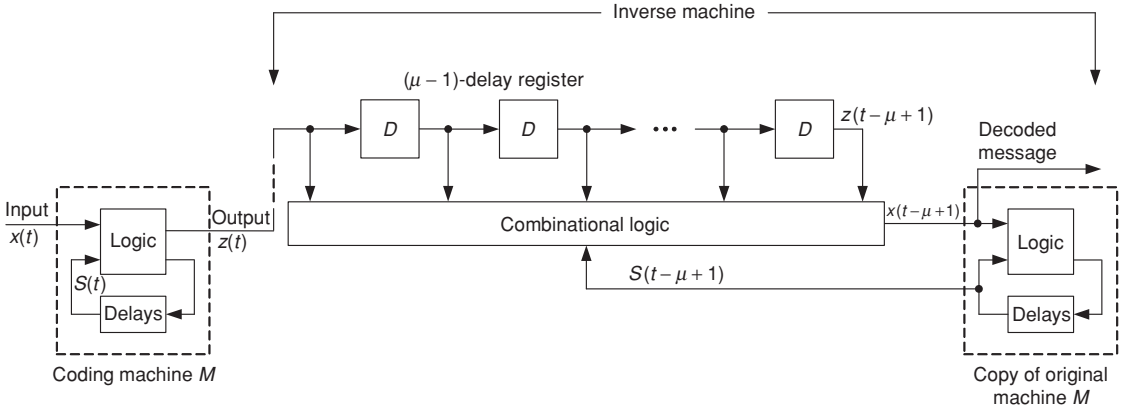


Fig. 14.12 Schematic diagram of a deciphering system.

Table 14.20 Machine M_{10}

	NS, z	
PS	$x = 0$	$x = 1$
A	$C, 0$	$D, 1$
B	$D, 0$	$C, 1$
C	$A, 0$	$B, 0$
D	$C, 1$	$D, 1$

The foregoing deciphering system does not yield an economical realization, since it requires a copy of the original machine as well as a $(\mu - 1)$-delay register. In fact, if we were to construct a composite state table for the inverse machine (i.e., a composite table for both the register and the copy of M), we would find that in many cases it can be considerably simplified. The question that now arises is whether we can find a minimal inverse directly from M's description, without going through the above construction procedure. Indeed, this can be accomplished, as will be shown subsequently.

*The minimal inverse machine

We shall demonstrate a construction procedure that yields a minimal inverse machine by finding the inverse of the machine M_{10} shown in Table 14.20. This machine is lossless of third order and, therefore, if we know the initial state and the values of three successive output symbols produced by transitions from this state then we can determine the first input symbol to the machine. Let us now define a set of triples, denoted $(S(t), z(t + 1), z(t + 2))$. The first member of each triple is a possible initial state of M_{10}; the second member is one of the output symbols that can be produced by a single transition from this state; and the third member is an output symbol that can follow this initial state and the first output symbol. A triple is defined for each possible initial state and for all possible output sequences of length 2. For the machine M_{10} we obtain the following triples:

$$\begin{array}{cccc} (A, 0, 0) & (B, 0, 1) & (C, 0, 0) & (D, 1, 0) \\ (A, 1, 1) & (B, 1, 0) & (C, 0, 1) & (D, 1, 1) \end{array}$$

The triple $(A, 0, 1)$, for example, is not defined because the output sequence 01 cannot be generated by M_{10} when it is initially in state A.

The set of triples so generated contains all possible combinations of initial states and output sequences of length 2. To determine the input symbol that causes the transition from the initial state while producing the output symbol specified by the second member of the triple, all that is necessary is one additional output symbol. Accordingly, if we construct a machine, each of whose states corresponds to a triple and represents the "information" carried

Table 14.21 Machine M_{10}^{i}

	NS, x	
PS	$z = 0$	$z = 1$
$(A, 0, 0)$	$(C, 0, 0), 0$	$(C, 0, 1), 0$
$(A, 1, 1)$	$(D, 1, 0), 1$	$(D, 1, 1), 1$
$(B, 0, 1)$	$(D, 1, 0), 0$	$(D, 1, 1), 0$
$(B, 1, 0)$	$(C, 0, 0), 1$	$(C, 0, 1), 1$
$(C, 0, 0)$	$(A, 0, 0), 0$	$(B, 0, 1), 1$
$(C, 0, 1)$	$(B, 1, 0), 1$	$(A, 1, 1), 0$
$(D, 1, 0)$	$(C, 0, 0), 0$	$(C, 0, 1), 0$
$(D, 1, 1)$	$(D, 1, 0), 1$	$(D, 1, 1), 1$

by that triple, and if we supply the machine with the output symbols of the original machine, then it will have all the necessary information to compute the input symbols in question.

The inverse of the machine M_{10}, denoted M_{10}^{i}, has eight states corresponding to the eight triples derived earlier. We shall often refer to a state of the inverse machine as an *inverse state*. For every state of M_{10}^{i}, the next inverse state is a triple whose members are obtained in the following manner.

1. The first member is the state to which machine M_{10} goes when it is initially in the state that is the first member of the present inverse state, and when it is supplied with the first input symbol.
2. The second member is the third member of the corresponding present inverse state.
3. The third member is the present output of M_{10}.

The state table of the machine M_{10}^{i} is given in Table 14.21. Suppose, for example, that M_{10}^{i} is in the state $(A, 0, 0)$ and that its current input symbol is 0. To obtain its 0-successor, we observe that M_{10}, when initially in state A, can produce three consecutive 0 output symbols only if the first input symbol is 0; as a result, M_{10}'s first transition is to state C and the 0-successor of $(A, 0, 0)$ contains C as its first member. The second member of the triple $(C, 0, 0)$ equals the third member of $(A, 0, 0)$, while its third member is the current output symbol of M_{10}, which constitutes the current input symbol to M_{10}^{i} and is given by M_{10}^{i}'s input column heading. The output sequence of M_{10}^{i} is a delayed replica of the input sequence to M_{10}; that is, the output symbol of M_{10}^{i} at t is equal to M_{10}'s input symbol at $t - 2$.

The set of states generated by the set of triples is clearly sufficient for a realization of the inverse machine. It does not, however, yield the smallest set of states. The machine M_{10}^{i}, for example, can be reduced since $(A, 0, 0)$ is equivalent to $(D, 1, 0)$ and similarly $(A, 1, 1)$ is equivalent to $(D, 1, 1)$.

Table 14.22 The minimal machine M_{10}^{i}

	NS, x	
PS	$z = 0$	$z = 1$
S_1	$S_5, 0$	$S_6, 0$
S_2	$S_1, 1$	$S_2, 1$
S_3	$S_1, 0$	$S_2, 0$
S_4	$S_5, 1$	$S_6, 1$
S_5	$S_1, 0$	$S_3, 1$
S_6	$S_4, 1$	$S_2, 0$

If we denote $(A, 0, 0)$ by S_1, $(A, 1, 1)$ by S_2, and so on, we obtain the minimal inverse, given in Table 14.22.

The foregoing procedure is applicable to any lossless machine of finite order. In general, for a machine of order μ we define a set of μ-tuples that constitutes the set of states of the inverse machine. The first member of each μ-tuple is a state of the original machine M; the remaining members are the possible output sequences of length $\mu - 1$ that can be produced by successive transitions from that state. The fact that this procedure yields more economical realizations than the "canonic" realization of the preceding section can be explained as follows. In the canonic realization, we stored the output sequence in a shift register and used a copy of the original machine to provide the information regarding the state of the original machine. In the present realization we use the same memory devices to store information regarding both the states and output sequences, thus achieving a reduction in the number of states of the inverse machine.

Suppose that M_{10} is initially in state A and, in response to some input sequence, it produces one of the output sequences 00 or 11. Then, two units of time later, M_{10}^{i} must be in the state that corresponds to A and the appropriate output sequence, i.e., $(A, 0, 0)$ or $(A, 1, 1)$. However, since $S_4 = (B, 1, 0)$ is the only state from which M_{10}^{i} can reach $(A, 0, 0)$ and $(A, 1, 1)$ when supplied with the input sequences 00 and 11 respectively, it follows that if the initial state of M_{10} is A then the initial state of M_{10}^{i} must be $(B, 1, 0)$. In a similar fashion, the reader can verify that if M_{10} is initially in state B then M_{10}^{i} can be initially in either S_1 or S_4 and if M_{10} is initially in either state C or D then M_{10}^{i} can be initially in S_2, S_3, S_5, or S_6.

As an example demonstrating the deciphering capability of M_{10}^{i} let M_{10} and M_{10}^{i} be initially in states A and S_4 respectively and let the input sequence 010001101 be applied to M_{10}. The deciphering process is shown in Fig. 14.13. The first two output symbols of M_{10}^{i}, as well as the last two input symbols to M_{10}, must be ignored. In the remaining positions of both sequences, the input to M_{10} and output of M_{10}^{i} are identical although shifted in time.

Table 14.23 A binary code

Source symbols	Code words
A	00
B	01
C	11
D	10

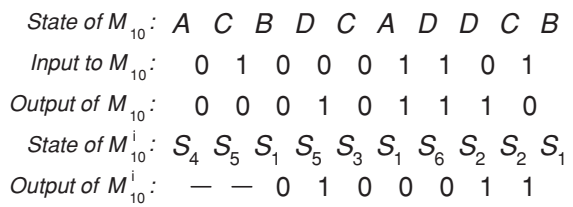


Fig. 14.13 Deciphering by means of M_{10}^{i}.

*14.5 Synchronizable and uniquely decipherable codes

The objective of this section is twofold: to introduce some of the basic issues in coding theory and to demonstrate the applicability of the preceding testing techniques to the area of information transmission and codes. We do not intend to develop the entire subject of coding theory but, rather, to illustrate some aspects of this subject that are relevant to the memory and information-losslessness aspects of automata. These concepts will, therefore, be introduced without formal definitions and proofs.

Introduction

Let the symbols $\{A, B, C, \ldots\}$ denote a finite *source alphabet*, and let $L = \{0, 1, 2, \ldots\}$ be a *code alphabet*. We shall be concerned only with binary codes, where $L = \{0, 1\}$. A concatenation of a finite number of code symbols is referred to as a *code word*. A *code* consists of a finite number of distinct code words of finite length, each representing a source symbol. A coded message is constructed by concatenating code words without spacing or any other punctuation. For example, let the code alphabet be $L = \{0, 1\}$ and the set of code words γ_1 be $\{00, 01, 11, 10\}$. The code shown in Table 14.23 is a mapping from the source alphabet $\{A, B, C, D\}$ to γ_1. Thus, the sequence $ABDC$ would be coded as 00011011.

By using the code in Table 14.23 we may obtain a sequence of binary digits for any sequence of source symbols. We may also work backward to obtain a sequence of source symbols for any sequence of binary digits arising from this code. In fact, since each source symbol is represented by a distinct code word and all code words are of equal length, to every sequence of code words from

this code there corresponds a unique sequence of source symbols. Not in every case can we work backward and find a unique sequence of source symbols that corresponds to a given binary sequence. For example, if $\gamma_2 = \{0, 00, 01\}$ is the code representing $\{A, B, C\}$ then the sequence 0001 may be decoded as either AAC or BC.

A code is said to be *uniquely decipherable* if and only if every coded message can be decomposed into a sequence of code words in only one way. Thus, γ_1 is uniquely decipherable while γ_2 is not. Whenever the number of code symbols is not the same for all code words the code is not necessarily uniquely decipherable, as illustrated by γ_2. However, the code $\gamma_3 = \{1, 01, 001, 0001\}$ is uniquely decipherable since the symbol 1 actually serves as a separator between successive code words. Such a separator is referred to as a *comma*, and such a code is called a *comma code*. A code in which all code words contain the same number of symbols is called a *block code*. A code in which the numbers of symbols representing different code words are not the same is called a *variable-length code*.

Whenever each code word can be deciphered without knowledge of the succeeding code words, the code is said to be an *instantaneous code*. For example, γ_1 and γ_3 are instantaneous codes while $\gamma_4 = \{1, 10, 100\}$ is not, since the sequence 10 cannot be deciphered until we verify that the next symbol is a 1.

Let $\xi = \xi_1\xi_2 \cdots \xi_n$ be a code word; then the sequence of code symbols $\xi_1\xi_2 \cdots \xi_m$, where $m \leq n$, is called a *prefix* of ξ. It can be shown that a necessary and sufficient condition for a code to be instantaneous is that no code word is a prefix of some other code word. Clearly, γ_4 is not instantaneous because 1 is a prefix of both 10 and 100.

A major reason for using variable-length codes is the consequent reduction in the average length of coded messages. Certain symbols of the source alphabet are more frequently used than others. For example, in English the letter e is more often used than the letter q. It is advantageous to assign shorter code words to those symbols that appear most often and longer code words to other symbols. If we let P_i and l_i denote, respectively, the probability of occurrence and the length of the code word representing the ith source symbol then we obtain the average length of the code, which is defined as the sum $\sum P_i l_i$ over all code words. For a given source alphabet and a given code alphabet, it is usually possible to construct many uniquely decipherable codes. In some codes, however, if an error occurs at the beginning of the coded message then it may invalidate the entire message. It is therefore desirable to have codes that are *synchronizable*, that is, for which the propagation of an error is bounded to a fixed portion of the message.

A test for unique decipherability

A code is said to be *uniquely decipherable with a finite delay* μ if and only if μ is the least integer such that knowledge of the first μ symbols of the coded

Table 14.24 Testing table for $\gamma = \{0, 01, 1010\}$

	0	1
S	(SB_1)	—
SB_1	—	(SC_1)
SC_1	$(SC_2)(B_1C_2)$	—
SC_2	—	(C_1C_3)
B_1C_2	—	(SC_3)
C_1C_3	(SC_2)	—
SC_3	$(SB_1)(SS)$	—

message suffices to determine its first code word. We now present a testing procedure to determine whether a code is uniquely decipherable and, if it is, the delay μ. This procedure is analogous to tests for information losslessness or for information losslessness of finite order.

Let us insert a separation symbol S at the beginning and end of each code word in γ. In addition, in every code word representing the source symbol N, we insert the symbol N_i between its ith symbol and its $(i+1)$th symbol. For example, if the source symbols are $\{A, B, C\}$ and $\gamma = \{0, 01, 1010\}$ then the code words with the inserted symbols are as follows:

$$\begin{array}{lllllllllll} A & \rightarrow & S & 0 & S & & & & & & \\ B & \rightarrow & S & 0 & B_1 & 1 & S & & & & \\ C & \rightarrow & S & 1 & C_1 & 0 & C_2 & 1 & C_3 & 0 & S \end{array}$$

Each code symbol ξ_k is now situated between two separation symbols. We say that the separation symbol to the right of the code symbol is the ξ_k-*successor*, denoted R_i, of the left separation symbol. For example, C_1 is the 1-successor of S because $S1C_1$ occurs in the third code word. Two successors, R_i and R_j, are *compatible* if $S\xi_k R_i$ and $S\xi_k R_j$ occur in the code words, or if $R_p\xi_k R_i$ and $R_q\xi_k R_j$ occur, and R_p and R_q are compatible. In such a case, $(R_i R_j)$ is said to be the compatible pair *implied* by $(R_p R_q)$.

A *testing table* (*for unique decipherability*) can now be constructed in the following manner.

1. The column headings of the table are the symbols of the code alphabet.
2. The first row heading is S. The other row headings are the compatible pairs.
3. The entries in row R_pR_q, column ξ_k, are the compatible pairs implied by (R_pR_q) under ξ_k.

The testing table for our example is given in Table 14.24. The entry in row S, column 0, is (SB_1), since $S0S$ and $S0B_1$ occur in the first and second words. The compatible implied by (SB_1) is (SC_1), since S is the 1-successor of B_1 in code word B while C_1 is the 1-successor of S in code word C; i.e., B_11S and $S1C_1$ occur in the code words. If $(R_iR_jR_k)$ is a compatible, we enter into the

Table 14.25 The testing table for $\gamma = \{1, 10, 001\}$

		0	1
$A \to S1S$	S	—	(SB_1)
$B \to S1B_10S$	(SB_1)	(SC_1)	—
$C \to S0C_10C_21S$	(SC_1)	(C_1C_2)	—
	(C_1C_2)	—	—

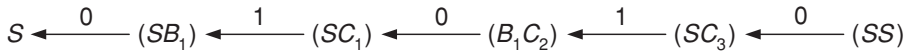


Fig. 14.14 Determination of an ambiguous message.

Fig. 14.15 Testing graph.

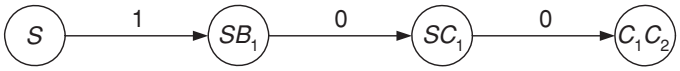


table all unordered pairs (R_iR_j), (R_iR_k), (R_jR_k). The table is complete when all the compatible pairs have been used as row headings.

If during the construction of the testing table a repeated pair (SS) occurs then the code is not uniquely decipherable. The occurrence of such a compatible pair means that there exists some compatible pair (R_iR_j) such that S is the ξ-successor of both R_i and R_j. However, since both R_i and R_j (like all compatible pairs) are reachable from S by a binary sequence that corresponds to two or more different sequences of source symbols, the code is not uniquely decipherable. Moreover, by tracing back the compatible pairs that implied the pair (SS), we can find one of the shortest ambiguous messages, which in our example is 01010, as shown in Fig. 14.14. The pair (SS) is written in the rightmost position, and its 0-predecessor is written in the next-left position, and so on. The sequence of arrow labels leading from S to (SS) is an ambiguous message. Indeed, 01010 may be interpreted as AC or as BBA.

It is easy to show that if pair (SS) is not generated then the code is uniquely decipherable. Hence, *a necessary and sufficient condition for a code to be uniquely decipherable is that a pair* (SS) *is not generated in the testing table*.

A *testing graph* (*for unique decipherability*) G can now be constructed as follows.

1. Corresponding to every row in the testing table, create a vertex in G.
2. Take directed arcs from each such vertex to the vertices corresponding to the implied compatible pairs.

The testing table for the code $\gamma = \{1, 10, 001\}$ is shown in Table 14.25. The corresponding testing graph is shown in Fig. 14.15. Since pair (SS) has not been generated in the testing table, the code is uniquely decipherable.

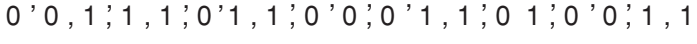


Fig. 14.16 Deciphering a coded message.

In analogy to Theorem 14.5, we can show that *a code is uniquely decipherable with finite delay μ if and only if its testing graph is loop-free*. The delay μ is equal to $l + 1$, where l is the length of the longest path in G. The longest path in the graph of Fig. 14.15 is 3 and thus $\mu = 4$.

Deciphering a coded message

We now describe a procedure to decipher a coded message. The decoding procedure is similar to the input-retrieval procedure for lossless machines and will be illustrated by means of an example. Consider the code $\gamma = \{11, 011, 001, 01, 00\}$, which is known to be uniquely decipherable, and suppose that we want to decode the sequence 0011101100011010011. Scanning the message from the left, we insert a lower comma whenever a sequence that corresponds to a legitimate code word is detected. For example, the first comma from the left follows the initial 00, since 00 is a code word in γ. Next, a comma follows the 1 since the sequence 001 is also a code word in γ, and so on. Although the tenth and eleventh symbols are 0's, no lower comma is inserted between the eleventh and twelfth symbols because there is no comma between the ninth and tenth symbols, and a new code word cannot start unless a comma indicates the end of the preceding code word. The procedure is illustrated in Fig. 14.16.

Next, we scan the coded message from the right and inset an upper comma whenever a sequence that corresponds to the inverse of a legitimate code word is scanned. The inverses of the code words in our example are $\{11, 110, 100, 10, 00\}$. If the code is uniquely decipherable then the message can be decoded by retaining only those commas that occur in the upper and lower spaces simultaneously. In our example, we find the following message:

$$001; 11; 011; 00; 011; 01; 00; 11$$

Although in general the above procedure will require keeping track of a number of sequences and the locations of the various commas, it is in principle a simple procedure that can be carried out by a finite-state machine.

A test for the synchronizability of codes

A code is said to be *synchronizable of order μ* if μ is the least integer such that the knowledge of any μ consecutive code symbols is sufficient to determine a separation of code words within these symbols. We shall restrict our attention to synchronizable codes that are uniquely decipherable with a finite delay, since these are the only ones of practical interest.

The problem of testing a code for synchronizability is analogous to the problem of testing a machine for finite output memory. In fact, since in both cases the objective is to specify the sequence at some point, we can use the same testing procedure. Let us construct a *testing table* (*for synchronizability*) in the following manner. The row headings in the upper half of the table consist of all the separation symbols. The column headings are the code symbols. The entries in row R_i, column ξ_k, of the upper half of the table are the ξ_k-successors of R_i. The row headings in the lower half of the table are all pairs of separation symbols. The entries in row R_iR_j, column ξ_k, are the pairs implied by (R_iR_j) and symbol ξ_k. The *testing graph* (*for synchronizability*) has a vertex for each row in the lower half of the testing table. A directed arc labeled ξ_k leads from the vertex R_iR_j to the vertex R_pR_q, where $p \neq q$, if and only if (R_pR_q) is the ξ_k-successor of (R_iR_j). We now state, without proof, the necessary and sufficient condition for a code to be synchronizable.

- A code is synchronizable if and only if it is uniquely decipherable and its testing graph is loop-free. It is synchronizable of order μ if and only if the longest path in the graph is of length $\mu - 1$.

Example Consider the code $\gamma = \{1, 10, 001\}$, whose testing table is shown in Table 14.26 and testing graph in Fig. 14.17. Since the code is uniquely decipherable and the graph is loop-free, γ is synchronizable of order 5.

Table 14.26 The testing table for $\gamma = \{0, 10, 001\}$

	0	1
S	C_1	(SB_1)
B_1	S	—
C_1	C_2	—
C_2	—	S
SB_1	(SC_1)	—
SC_1	(C_1C_2)	—
SC_2	—	(SB_1)
B_1C_1	(SC_2)	—
B_1C_2	—	—
C_1C_2	—	—

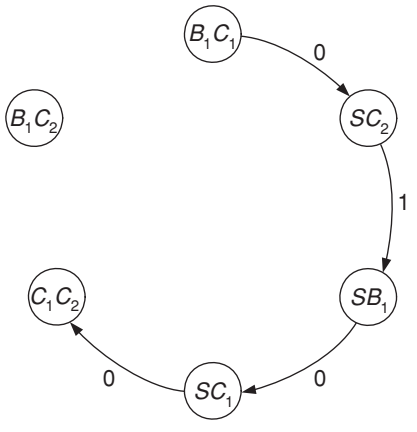


Fig. 14.17 Testing graph.

The main advantage of using a synchronizable code is that the propagation of errors within messages composed of such a code is bounded. In other words, if an error occurs in transmitting a coded message, its effect on the decipherability of the message is limited to at most μ symbols, since the knowledge of any μ code symbols is sufficient to determine a single separation within these symbols.

Table 14.27 State table of an information lossless machine of maximal order

	NS, z			
PS	I_1	I_2	I_3	I_4
1	$2, 0$	$3, 2$	$2, 3$	$2, 5$
2	$3, 0$	$4, 2$	$3, 3$	$3, 5$
3	$4, 0$	$5, 2$	$4, 3$	$4, 5$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	$i+1, 0$	$i+2, 2$	$i+1, 3$	$i+1, 5$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\lceil \frac{1}{2}n \rceil - 1$	$\lceil \frac{1}{2}n \rceil, 0$	$\lceil \frac{1}{2}n \rceil + 1, 2$	$\lceil \frac{1}{2}n \rceil, 3$	$\lceil \frac{1}{2}n \rceil, 5$
$\lceil \frac{1}{2}n \rceil$	$\lceil \frac{1}{2}n \rceil + 1, 0$	$\lceil \frac{1}{2}n \rceil + 2, 1$	$\lceil \frac{1}{2}n \rceil + 1, 3$	$\lceil \frac{1}{2}n \rceil + 1, 5$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
j	$j+1, 0$	$j+2, 1$	$j+1, 3$	$j+1, 5$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n-2$	$n-1, 0$	$n, 1$	$n-1, 3$	$n-1, 5$
$n-1$	$n, 0$	$1, 1$	$1, 6$	$n, 5$
n	$1, 4$	$1, 2$	$n, 3$	$2, 4$

In addition, since synchronizable codes are also uniquely decipherable with a finite delay, the determination of a single separation of code words is sufficient for the decoding of the message from that point on.

*Appendix 14.1 The least upper bound for information losslessness of finite order

In the following, we shall prove that the bound for information losslessness established by Theorem 14.5 is the least upper bound. Specifically, we shall show that, for every n, there exists a machine with four input symbols and seven output symbols which is information lossless of maximal order, that is, for which $\mu = 1 + \frac{1}{2}(n-1)n$. Such a machine is shown in Table 14.27, where $\lceil g \rceil$ is the least integer greater than or equal to g.

Theorem 14.6 *For every n there exists an information lossless machine of order*

$$\mu = 1 + \frac{(n-1)n}{2}.$$

Proof We prove the theorem by demonstrating that the class of machines described in Table 14.27 is information lossless of order $1 + \frac{1}{2}(n-1)n$. The upper part of the testing table for this machine is given in Table 14.28. The testing graph is derived directly from the table and is shown for even n in

Table 14.28 Testing table for information losslessness for the machine in Table 14.27

	Output						
PS	0	1	2	3	4	5	6
1	2	—	3	2	—	2	—
2	3	—	4	3	—	3	—
3	4	—	5	4	—	4	—
4	5	—	6	5	—	5	—
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$\lceil \frac{1}{2}n \rceil - 1$	$\lceil \frac{1}{2}n \rceil$	—	$\lceil \frac{1}{2}n \rceil + 1$	$\lceil \frac{1}{2}n \rceil$	—	$\lceil \frac{1}{2}n \rceil$	—
$\lceil \frac{1}{2}n \rceil$	$\lceil \frac{1}{2}n \rceil + 1$	$\lceil \frac{1}{2}n \rceil + 2$	—	$\lceil \frac{1}{2}n \rceil + 1$	—	$\lceil \frac{1}{2}n \rceil + 1$	—
$\lceil \frac{1}{2}n \rceil + 1$	$\lceil \frac{1}{2}n \rceil + 2$	$\lceil \frac{1}{2}n \rceil + 3$	—	$\lceil \frac{1}{2}n \rceil + 2$	—	$\lceil \frac{1}{2}n \rceil + 2$	—
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$n-3$	$n-2$	$n-1$	—	$n-2$	—	$n-2$	—
$n-2$	$n-1$	n	—	$n-1$	—	$n-1$	—
$n-1$	n	1	—	—	—	n	1
n	—	—	1	n	(1, 2)	—	—

Fig. 14.18. The graph contains no vertex with repeated entries because all the entries in every column of the upper part of the testing table are distinct. The graph contains $\frac{1}{2}(n-1)n$ vertices arranged in $n-1$ columns. The maximal path, which connects all these vertices, is shown in Fig. 14.18 by the solid lines. The maximal path is constructed in the following manner. The first compatible pair (1, 2) is introduced in column 4 of the testing table. This pair, in turn, implies the pairs (2, 3), (3, 4), ..., $(n-2, n-1)$. Because of the arrangement of the entries in column 1 of Table 14.28, the pair $(1, n)$ is implied by $(n-2, n-1)$. In addition, because of the entries in column 2, the pair (1, 3) is implied by $(1, n)$ and similarly for every column of vertices in the graph. The path goes from the vertex $(1, k)$, for all $2 \leq k \leq \frac{1}{2}n$, to the vertex $(n-k, n-1)$, from which it goes to the vertex $(1, n-k+2)$, as implied by the entries in column 1 of the testing table.

The path continues from the vertex $(1, h)$, for all $(\frac{1}{2}n) + 1 < h \leq n$ to the vertex $(n-h+1, n)$, from which it goes to $(1, n-h+3)$, as implied by the entries in column 2 of the testing table. Finally, the path goes from the vertex $(\frac{1}{2}n, n)$ to $(\frac{1}{2}n+1, n)$, and so on to $(n-1, n)$, as implied by entries in column 3 of Table 14.28. The vertex $(n-1, n)$ is a terminal vertex since the corresponding compatible pair implies no other compatible pair.

It is evident from the structure of the graph that it has no loops, although it contains a number of shorter paths. The testing graph for n odd can be obtained from Table 14.28 in a similar manner, and it too has a path that connects all $\frac{1}{2}n(n-1)$ vertices. Consequently, for any given n the machine in Table 14.27 is information lossless of maximal order. ◇

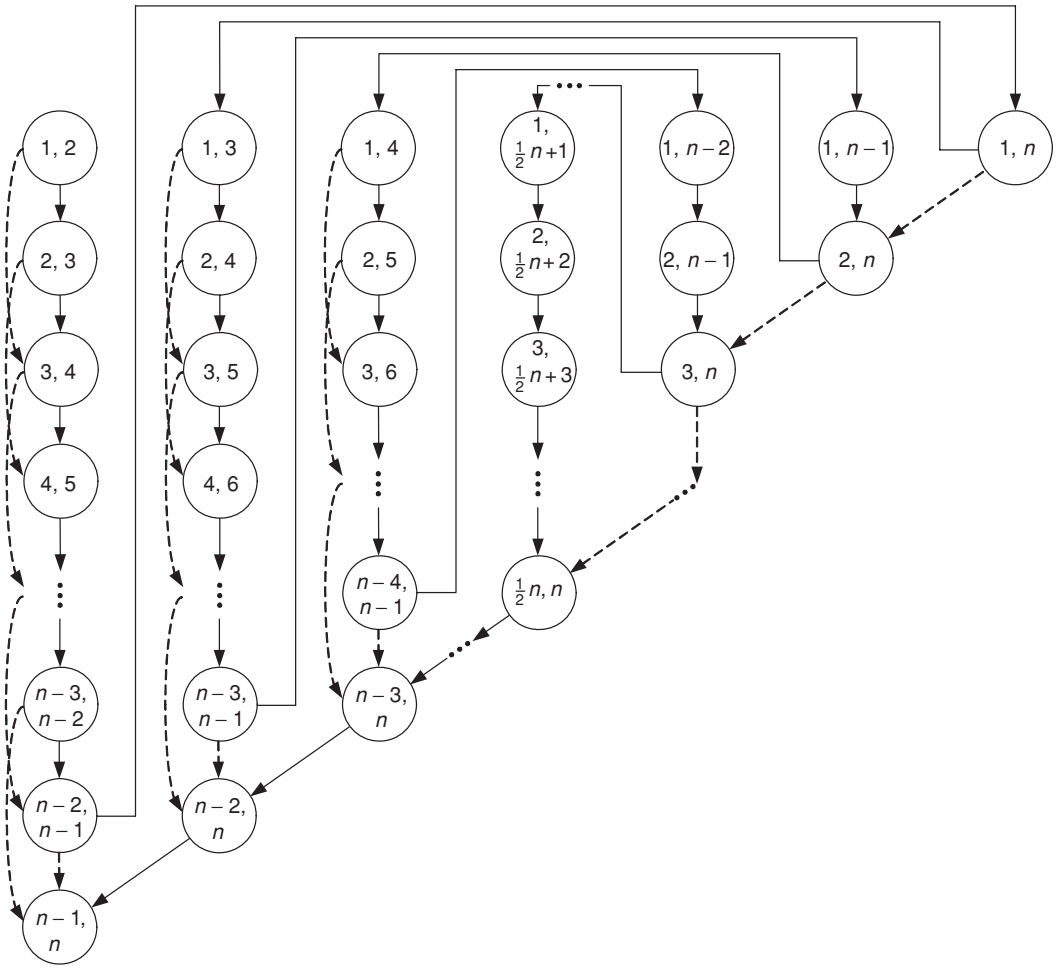


Fig. 14.18 Testing graph for even n for the lossless machine of Table 14.27.

It seems that it may be possible to find an information lossless machine of maximal order with fewer inputs or outputs. It is not clear, however, whether there exists such a machine with only two input symbols and two output symbols.

Notes and references

The various memory aspects of automata have been investigated by numerous authors, among whom are Liu [6, 7, 8], McCluskey [10], Massey [9], Simon [12], and Perles, Rabin, and Shamir [11]. Lossless machines were first studied by Huffman [4], who devised tests for losslessness and losslessness of finite order. Even [1] devised a different testing procedure, the one adopted in this chapter. The least upper bound developed in the above appendix is due to Kohavi and Winograd [5]. The tests for decipherability and synchronizability of codes are due to Even [2, 3].

[1] Even, S.: "On information lossless automata of finite order," *IEEE Trans. Electron. Computers*, vol. EC-14, pp. 561–569, August 1965.
[2] Even, S.: "Test for synchronizability of automata and variable length codes," *IEEE Trans. Information Theory*, vol. IT-10, pp. 185–189, July 1964.
[3] Even, S.: "Tests for unique decipherability," *IEEE Trans. Information Theory*, vol. IT-9, pp. 109–112, April 1963.
[4] Huffman, D. A.: "Canonical forms for information lossless finite-state machines," *IRE Trans. Circuit Theory*, vol. CT-6, Special Supplement, pp. 41–59, May 1959.
[5] Kohavi, Z., and J. Winograd: "Establishing bounds concerning finite automata," *J. Computer and System Sciences*, vol. 7, no. 3, pp. 288–299, June 1973.
[6] Liu, C. L.: "Some memory aspects of finite automata," MIT. Res. Lab. Electron. Tech. Rept 411, May 1963.
[7] Liu, C. L.: "Determination of the final state of an automaton whose initial state is unknown," *IEEE Trans. Electron. Computers*, vol. EC-12, December 1963.
[8] Liu, C. L.: "*k*th-order finite automaton," *IEEE Trans. Electron. Computers*, vol. EC-12, October 1963.
[9] Massey, J. L.: "Note on finite-memory sequential machines," *IEEE Trans. Electron. Computers*, vol. EC-l5, pp. 658–659, 1966.
[10] McCluskey, E. J.: "Reduction of feedback loops in sequential circuits and carry leads in iterative networks," in *Proc. Third Ann. Symp. Switching Theory and Logical Design*, pp. 91–102, Chicago, 1962.
[11] Perles, M., M. O. Rabin, and E. Shamir: "The theory of definite automata," *IEEE Trans. Electron. Computers*, pp. 233–243, June 1963.
[12] Simon, S. M.: "A note on memory aspects of sequence transducers," *IRE Trans. Circuit Theory*, vol. CT-6, Special Supplement, pp. 26–29, May 1959.

Problems

Problem 14.1. For each of the machines in Table P14.1, determine whether it has a finite memory and, if it does, find its order.

Table P14.1

	NS, z	
PS	$x=0$	$x=1$
A	$B,0$	$B,0$
B	$C,0$	$D,0$
C	$D,0$	$C,0$
D	$A,0$	$C,1$

(*a*)

	NS, z	
PS	$x=0$	$x=1$
A	$D,0$	$C,1$
B	$A,0$	$E,0$
C	$C,1$	$E,0$
D	$C,1$	$C,1$
E	$B,0$	$B,1$

(*b*)

	NS, z	
PS	$x=0$	$x=1$
A	$B,0$	$E,0$
B	$C,0$	$D,0$
C	$D,0$	$C,0$
D	$E,0$	$A,0$
E	$E,0$	$A,1$

(*c*)

Problem 14.2. The *canonical realization of finite-memory machines* is shown in Fig. P14.2. Verify that the machine of Table P14.2 has a finite memory, and show its canonical realization. In particular, design the combinational logic.

Table P14.2

	NS, z	
PS	$x = 0$	$x = 1$
A	$A, 0$	$B, 1$
B	$C, 0$	$D, 1$
C	$B, 1$	$A, 0$
D	$D, 1$	$C, 0$

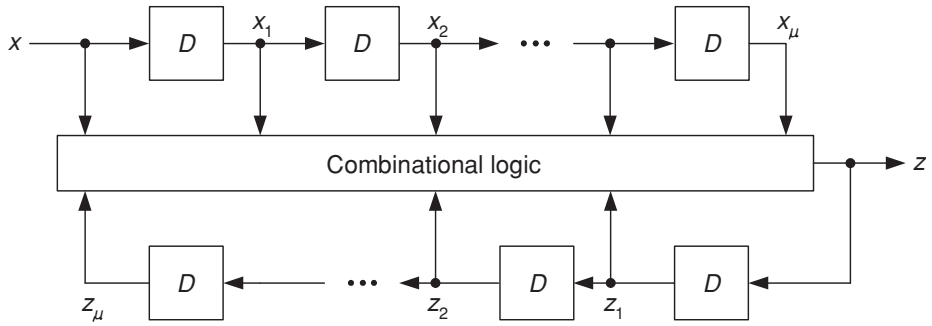


Fig. P14.2

Problem 14.3. Prove that, for every n, the machine of Table P14.3 has a finite memory of order $\mu = \frac{1}{2}(n-1)n$. (Recall that $\lceil g \rceil$ is the least integer greater than or equal to g.)

Hint: Use a testing graph for finite memory.

Table P14.3

	NS		z	
PS	$x = 0$	$x = 1$	$x = 0$	$x = 1$
1	2	3	0	0
2	3	4	0	0
3	4	5	0	0
4	5	6	0	0
⋮	⋮	⋮	⋮	⋮
$\lceil \frac{1}{2}(n-3) \rceil$	$\lceil \frac{1}{2}(n-1) \rceil$	$\lceil \frac{1}{2}(n+1) \rceil$	0	0
$\lceil \frac{1}{2}(n-1) \rceil$	$\lceil \frac{1}{2}(n+1) \rceil$	$\lceil \frac{1}{2}(n+3) \rceil$	0	1
$\lceil \frac{1}{2}(n+1) \rceil$	$\lceil \frac{1}{2}(n+3) \rceil$	$\lceil \frac{1}{2}(n+5) \rceil$	0	1
⋮	⋮	⋮	⋮	⋮
$n-3$	$n-2$	$n-1$	0	1
$n-2$	$n-1$	n	0	1
$n-1$	n	1	0	1
n	n	1	0	0

Problem 14.4. Let M be a p-input symbol, q-output symbol, n-state, strongly connected machine. Prove that if M has a finite memory of order μ then $(pq)^{\mu} \geq n$.

Problem 14.5

(a) Test the machine of Table P14.5 for definiteness.

(b) Show the canonical realization of this machine (see Fig. 14.3). In particular, specify the combinational logic.

Table P14.5

	NS, z	
PS	$x = 0$	$x = 1$
A	$D, 1$	$E, 0$
B	$A, 0$	$B, 1$
C	$C, 0$	$B, 0$
D	$C, 1$	$B, 1$
E	$A, 0$	$B, 0$

Problem 14.6

(a) Specify the unspecified entries in Table P14.6*a* in such a way that the resulting machine will be definite. Is your answer unique? If not, show all possible ways to specify the table.

(b) Is it possible to specify Table P14.6*b* in such a way that it corresponds to a definite machine? Justify your answer.

Table P14.6

	NS	
PS	$x = 0$	$x = 1$
A	A	B
B	—	B
C	E	—
D	—	F
E	—	D
F	E	—

(*a*)

	NS	
PS	$x = 0$	$x = 1$
A	A	B
B	C	C
C	—	—
D	—	—

(*b*)

Problem 14.7. Determine which of the machines in Table P14.7 has a finite output memory, and find its order.

Table P14.7

	NS, z	
PS	$x = 0$	$x = 1$
A	$A, 0$	$B, 1$
B	$C, 1$	$D, 0$
C	$D, 0$	$C, 1$
D	$B, 1$	$A, 0$

(*a*)

	NS, z	
PS	$x = 0$	$x = 1$
A	$C, 0$	$C, 0$
B	$D, 1$	$A, 0$
C	$C, 1$	$B, 0$
D	$D, 1$	$D, 1$

(*b*)

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 0$	$C, 0$
B	$D, 0$	$E, 1$
C	$F, 1$	$D, 0$
D	$F, 1$	$F, 1$
E	$B, 0$	$B, 0$
F	$A, 1$	$A, 1$

(*c*)

Problem 14.8. Given the state table of the machine M shown in Table P14.8, specify the missing output entries in such a way that the machine will be finite-memory of maximal order.

Table P14.8

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 0$	$C, 1$
B	$D, 0$	$D, -$
C	$C, -$	$A, 0$
D	$C, 0$	$A, 1$

Problem 14.9. Given a machine M with n states $S_1, S_2, \ldots, S_n$:

(a) Devise a procedure to determine whether the machine has n preset sequences $X_1, X_2, \ldots, X_n$ such that X_i is the shortest sequence that takes M from any unknown initial state to state S_i.

(b) Apply your procedure to find the appropriate sequences for the machine M in Table P14.9.

(c) Find an upper bound on the length of X_i.

(d) Does the existence of such a set of sequences imply that M must be a definite machine?

Table P14.9

	NS, z	
PS	$x = 0$	$x = 1$
A	$C, 0$	$B, 0$
B	$E, 1$	$F, 0$
C	$A, 1$	$F, 1$
D	$E, 0$	$B, 1$
E	$C, 1$	$D, 0$
F	$E, 0$	$F, 0$

Problem 14.10. Consider the class of machines that have a finite output memory of order μ such that knowledge of the last μ output symbols suffices to determine the final state of the machine.

(a) Devise a test to determine whether a given machine belongs to the above class.

(b) Find such a four- or five-state machine and apply your test to it.

Problem 14.11. For each machine in Table P14.11, determine whether it is lossless. If it is lossy, find a shortest output sequence produced by two different input sequences with the same initial and final states. If it is lossless, determine its order.

Table P14.11

	NS, z	
PS	$x=0$	$x=1$
A	$B,1$	$C,0$
B	$A,0$	$D,1$
C	$B,0$	$A,0$
D	$C,1$	$A,1$

(a)

	NS, z	
PS	$x=0$	$x=1$
A	$B,0$	$C,1$
B	$D,1$	$A,0$
C	$E,1$	$F,1$
D	$F,0$	$E,0$
E	$C,1$	$A,0$
F	$B,0$	$D,1$

(b)

	NS, z	
PS	$x=0$	$x=1$
A	$B,0$	$C,0$
B	$D,0$	$E,1$
C	$E,0$	$A,1$
D	$E,0$	$D,0$
E	$C,1$	$B,1$

(c)

	NS, z	
PS	$x=0$	$x=1$
A	$B,0$	$A,1$
B	$C,0$	$D,1$
C	$E,1$	$A,0$
D	$E,0$	$C,0$
E	$C,1$	$E,0$

(d)

Problem 14.12. In Table P14.12 you are presented with only the lower half of a testing table (for losslessness) of an unknown machine. Specify the upper half of the table and find a corresponding four-state machine. Is your answer unique?

Table P14.12

	$z=0$	$z=1$
A		
B		
C		
D		
AB	—	$(BC)(CC)$
AC	—	$(AB)(AC)$
AD	—	—
BC	(BD)	(AC)
BD	$(AD)(CD)$	—
CD	$(AB)(BC)$	—

Problem 14.13

(a) The machine described in Table P14.13 has two binary outputs, z_1 and z_2. Some output entries are incompletely specified. Specify all these output entries in such a way that the machine will be lossless of first order.

(b) Prove that any binary-input binary-output machine can be transformed into a lossless machine of first order by adding to it a single binary output terminal.

Table P14.13

	NS, z_1z_2	
PS	$x=0$	$x=1$
A	$B,-1$	$C,11$
B	$D,0-$	$D,0-$
C	$D,0-$	$E,--$
D	$B,0-$	$D,-0$
E	$C,0-$	$D,-0$

Problem 14.14. The machine described in Table P14.14 has two binary outputs, z_1 and z_2, some of whose entries are incompletely specified. Specify all these output entries in such a way that the machine will be lossless of the least order. Is such a specification unique?

Table P14.14

	NS, z_1z_2	
PS	$x=0$	$x=1$
A	$B,10$	$C,10$
B	$C,00$	$C,1-$
C	$A,1-$	$D,00$
D	$D,1-$	$A,00$

Problem 14.15. Prove that the machine of Table P14.15 is lossless of maximal order, i.e., $\mu = 11$.

Table P14.15

	NS, z	
PS	$x=0$	$x=1$
S_1	$S_2,1$	$S_1,1$
S_2	$S_3,1$	$S_5,3$
S_3	$S_4,1$	$S_4,3$
S_4	$S_5,1$	$S_3,2$
S_5	$S_1,2$	$S_1,3$

Problem 14.16. For the machine shown in Table P14.16:

(a) Find in a systematic way output sequence Z_2 when output sequence Z_1 is 001001, and it is known that the initial and final states are both B.

(b) Given the initial and final states as well as output sequence Z_1, is it always possible to determine the output sequence Z_2?

Table P14.16

	NS, z_1z_2	
PS	$x=0$	$x=1$
A	$A, 11$	$B, 10$
B	$D, 00$	$A, 00$
C	$E, 00$	$C, 10$
D	$B, 01$	$C, 01$
E	$C, 11$	$A, 01$

Problem 14.17. For the machine shown in Table P14.17:
(a) Does the machine have a finite output memory? If yes, find the order λ.
(b) Is the machine information lossless of finite order? If yes, find the order μ.
(c) The machine produced an output sequence $Z = 0101000$. What is the corresponding input sequence? Is it unique?
(d) What is the minimal length of output sequence Z that enables us to determine at least one input symbol?

Table P14.17

	NS, z	
PS	$x=0$	$x=1$
A	$B, 0$	$C, 0$
B	$D, 0$	$E, 1$
C	$A, 1$	$E, 0$
D	$E, 0$	$D, 0$
E	$A, 1$	$E, 1$

Problem 14.18. Given the cascade connection of machines M_1 and M_2, as shown in Fig. P14.18:
(a) For M_1 and M_2 as shown in Table P14.18, given that the output sequence $Z = 110011$ and the final state of M_2 is B, determine the initial state of M_1.
(b) For the machines in Table P14.18 prove that, for every given output sequence Z of length L, knowledge of the final state of M_2 is sufficient to determine the state of M_1 at some time during the experiment. Find the value of L.

Fig. P14.18

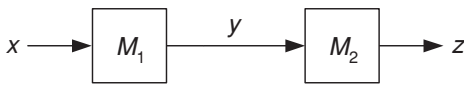


Table P14.18

	NS, y	
PS	$x=0$	$x=1$
A	$B, 0$	$C, 1$
B	$C, 0$	$B, 1$
C	$D, 0$	$D, 0$
D	$D, 0$	$A, 1$

	NS, z	
PS	$y=0$	$y=1$
A	$B, 0$	$C, 1$
B	$A, 0$	$C, 0$
C	$D, 1$	$A, 1$
D	$B, 1$	$D, 0$

Problem 14.19. The machines M_1 and M_2 shown in Table P14.19 are connected in cascade, as shown in Fig. P14.18. The initial state of M_1 is A. Find in a systematic way all the shortest input sequences which, when applied to M_1, make it possible to identify the initial state of M_2 by means of its response z.

Table P14.19

	NS, y	
PS	$x=0$	$x=1$
A	$B, 0$	$C, 1$
B	$C, 1$	$A, 0$
C	$A, 0$	$B, 0$

M_1

	NS, z	
PS	$y=0$	$y=1$
D	$E, 1$	$D, 0$
E	$F, 1$	$G, 0$
F	$D, 1$	$E, 0$
G	$F, 0$	$D, 0$

M_2

Problem 14.20

(a) In response to an unknown input sequence, the machine of Table P14.20 produces the output sequence 10011. Find the input sequence if it is known that the *final* state is B.

(b) Prove that knowledge of the final state of this machine and the last output symbol is sufficient to determine the next-to-final state.

(c) Devise a test, to determine whether a given machine is lossless, such that the knowledge of the *final* state and the last μ output symbols is sufficient to identify the next-to-final state.
Hint: Use the output-predecessor table.

Table P14.20

	NS, z	
PS	$x=0$	$x=1$
A	$B, 0$	$C, 1$
B	$A, 0$	$C, 0$
C	$D, 1$	$A, 1$
D	$B, 1$	$D, 0$

Problem 14.21

(a) In response to an unknown input sequence, the machine of Table P14.21 produces the output sequence 1110000010. Find the input sequence to the machine if it is known that its initial state is A and final state is F.

(b) Can the machine produce the output sequence 11011000 when both its initial and final states are A?

Table P14.21

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 1$	$C, 0$
B	$D, 1$	$B, 1$
C	$E, 1$	$B, 0$
D	$A, 0$	$E, 0$
E	$F, 0$	$D, 1$
F	$D, 0$	$A, 1$

Problem 14.22. Find a reduced four-state machine that is lossless of first order and is isomorphic to its own inverse.

Problem 14.23. Design an inverse of the machine shown in Table P14.23. Give a reduced, standard-form, state table, assuming that the initial state of the lossless machine is A. For each of the other possible initial states of this machine, specify appropriate initial states of the inverse.

Table P14.23

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 1$	$C, 1$
B	$D, 0$	$E, 0$
C	$A, 1$	$F, 1$
D	$C, 0$	$B, 0$
E	$F, 1$	$A, 1$
F	$E, 0$	$D, 0$

Problem 14.24

(a) Prove that the inverse of a lossless machine of finite order is a lossless machine of finite order.

(b) Demonstrate, by finding the inverse of the machine M_{10}^{i} (Table 14.22), that the inverse of the inverse of a lossless machine of finite order is isomorphic to the original machine, i.e., show that the inverse of M_{10}^{i} is isomorphic to M_{10}.

Problem 14.25. The output symbol of a finite-state machine is the modulo-2 sum of the current input symbol and the second and third past input symbols, i.e.,

$$z(t) = x(t) \oplus x(t-2) \oplus x(t-3).$$

(a) Prove that such a machine is lossless of finite order.
(b) Realize the machine and its inverse.

Problem 14.26. Show that the code $\gamma = \{1, 110, 010, 100\}$ is uniquely decipherable. Is it also uniquely decipherable with a finite delay? If so, find the delay; if not, show a message that cannot be deciphered in a finite time.

Problem 14.27. Given the uniquely decipherable code $\gamma = \{0, 001, 101, 011\}$, decipher the message 00101001101001100001.

CHAPTER

15 Linear sequential machines

Linear sequential machines constitute a subclass of linear systems in which the input vector, output vector, and state transitions occur in discrete steps. Consequently, the tools and techniques available for the analysis and synthesis of linear systems can be applied to linear machines as well. The numerous applications of linear machines give further incentive to the investigation of their properties and to the development of efficient synthesis procedures.

In the first few sections we present an intuitive, though well-justified, approach that requires only a limited knowledge of modem algebra. In subsequent sections (i.e., Sections 15.4 through 15.6) a matrix formulation is presented, and methods for minimizing and detecting linear machines are developed.

15.1 Introduction

A *linear sequential machine* (also called a *linear machine*) is a network that has a finite number of input and output terminals and is composed of interconnections of three types of basic components, to be introduced shortly. The input signals applied to the machine are elements of a finite field[1] $GF(p) = \{0, 1, \ldots, p-1\}$, and the operations performed by the basic components on their inputs are carried out according to the rules of $GF(p)$. A block-diagram representation of a linear machine with l input terminals and m output terminals is shown in Fig. 15.1.

For a machine to be linear, its response to a linear combination of inputs must preserve the scale factor and the principle of superposition. Thus, each of the basic components used to realize a linear machine must be linear. This requirement clearly precludes the use of an AND gate whose output is the product of its inputs; e.g., if the inputs are x_1 and x_2 and the signal values

[1] Some relevant basic properties of finite fields are summarized in Appendix 15.1. The understanding of these properties is essential to the study of linear machines.

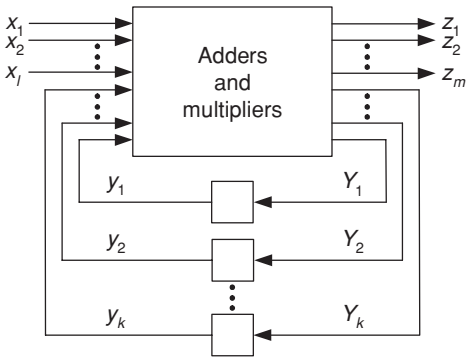


Fig. 15.1 Block diagram of a linear machine.

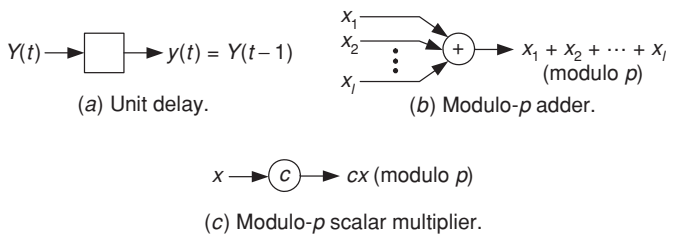


Fig. 15.2 Basic components of linear circuits.

are elements of $GF(2)$ then the output is $z = x_1x_2$ modulo 2. Using similar arguments, we observe that the OR gate is not linear either since, for example, the output[2] of a two-input gate is $z = x_1 + x_2 + x_1x_2$ modulo 2. The following three types of basic component are clearly linear.

1. *Unit delays* A unit delay is a two-terminal element whose output $y(t)$ is related to its input $Y(t)$ by $y(t) = Y(t-1)$.
2. *Modulo-p adders* An adder has l input terminals and one output terminal. The output is the modulo-p sum of the inputs; i.e., if the inputs are $x_1, x_2, \ldots, x_l$ then the output is $x_1 + x_2 + \cdots + x_l$ (modulo p).
3. *Modulo-p scalar multipliers* A multiplier c (where c is an element of $GF(p)$) has one input and one output terminal. If the input is x then the output is cx (modulo p).

Modulo-p addition and scalar multiplication are assumed to be executed instantaneously. For most purposes, we shall restrict p to prime numbers. The symbols representing the above components are shown in Fig. 15.2.

Any network that is constructed by interconnecting components of the types shown in Fig. 15.2 is referred to as a linear circuit, provided that every closed loop contains at least one delay element. The unit delay is equal to the discrete

[2] In this chapter, the symbol + represents the addition operation in accordance with the rules of $GF(p)$ (i.e., modulo p).

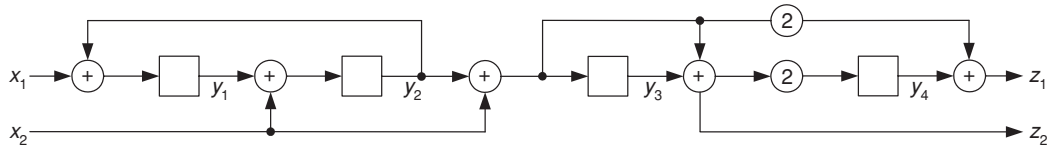


Fig. 15.3 A four-terminal four-dimensional linear machine over $GF(3)$.

interval of time between two successive clock pulses. The state variables of a linear machine are the outputs $y_1, y_2, \ldots, y_k$ of the delay elements. The state of a machine at time t is specified by the value of the y's at t, i.e., $y_1(t), y_2(t), \ldots, y_k(t)$. The number of delay elements (or state variables) in a linear machine is referred to as the *dimension* of the machine. A linear machine whose components are modulo p and whose input signals are elements of $GF(p)$ is said to be a linear machine *over GF(p)*.

Example Figure 15.3 illustrates a four-terminal four-dimensional linear machine over $GF(3)$.

A linear machine over $GF(2)$ is called a *binary* machine. Binary machines are practical and simple to construct and are widely used in various applications. Consequently, although we shall develop the theory of linear machines over $GF(p)$, most examples will be selected from linear machines over the $GF(2)$ field.

15.2 Inert linear machines

A linear machine whose delay elements are initially in the zero state is referred to as an *inert* (or *quiescent*) *linear machine*. Inert linear machines are used extensively as encoding and decoding devices and in various applications that require transformations of sequences. It will subsequently be shown that the study of these machines provides insight into the problem of arbitrary linear machines, as well as some of the basic tools for the analysis of the subject.

Feedforward shift registers

The simplest type of inert linear machine is a two-terminal shift register that contains only feedforward paths and whose output is a modulo-p sum of selected input digits. The schematic representation of a feedforward shift register over $GF(p)$ is shown in Fig. 15.4.

The output z can be described by a polynomial in D over the $GF(p)$ field, i.e.,

$$z = a_0 x + a_1 Dx + \cdots + a_k D^k x \tag{15.1}$$

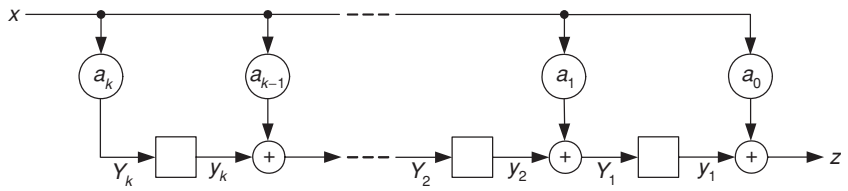


Fig. 15.4 A feedforward shift register.

where the symbol D^i is an i-unit *delay operator*, which delays by i time units the variable on which it operates. For example, equation $z = D^2x$ means that, for all $t \geq 2$, $z(t) = x(t-2)$. The operator $D^0 = 1$ is referred to as the *identity operator*. Equation (15.1) is a valid description of the shift register of Fig. 15.4 only if initial conditions of delays are zero, i.e., $y_1(0) = y_2(0) = \cdots = y_k(0) = 0$, since otherwise the output cannot be expressed for all times as only a function of the input. Equation (15.1) can be rewritten as

$$z = (a_0 + a_1 D + \cdots + a_k D^k)x$$

or as

$$\frac{z}{x} = a_0 + a_1 D + \cdots + a_k D^k = T(D), \tag{15.2}$$

where the polynomial $T(D)$, which expresses the ratio z/x, is defined as the *transfer function* of the inert linear machine.

Example Consider the inert linear machine over $GF(2)$ of Fig. 15.5, where the output digit is a modulo-2 sum of the present input digit and the first and third past input digits, i.e., $z(t) = x(t) + x(t-1) + x(t-3)$. The corresponding polynomial in the delay operator is

$$z = x + Dx + D^3x$$

and the transfer function is

$$T_1 = \frac{z}{x} = 1 + D + D^3.$$

Note that, for $GF(2)$, the scalar multiplier a_i is either 1 or 0, depending on whether there is or is not a connection to the ith modulo-2 adder.

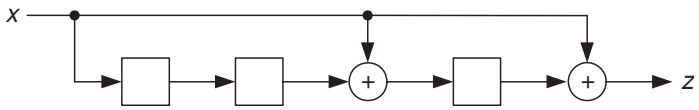


Fig. 15.5 Realization of the transfer function $T_1 = 1 + D + D^3$.

To show that the circuit represented by Eq. (15.1) and Fig. 15.4 is indeed linear, let z and z^* be the responses to two distinct input sequences x and x^*

respectively and let v and v^* be scalars taken from $GF(p)$. Then

$$z = a_0x + a_1Dx + \cdots + a_kD^kx$$

and

$$z^* = a_0x^* + a_1Dx^* + \cdots + a_kD^kx^*.$$

The response Z to a linear combination of inputs is given by

$$Z = a_0(vx + v^*x^*) + a_1D(vx + v^*x^*) + \cdots + a_kD^k(vx + v^*x^*)$$

or

$$Z = v(a_0x + a_1Dx + \cdots + a_kD^kx) + v^*(a_0x^* + a_1Dx^* + \cdots + a_kD^kx^*).$$

Hence,

$$Z = vz + v^*z^*. \tag{15.3}$$

The response of the machine to a linear combination of inputs preserves the scale factor and principle of superposition and consequently the machine is linear. As a result, we may apply the linear theory of polynomials to delay polynomials as well.

Consider now a serial connection of two linear machines of the type shown in Fig. 15.4; that is, the output of the predecessor machine is the input to the successor machine. Let x_1, z_1, and T_1 denote the input, output, and transfer function of the predecessor machine, and let x_2, z_2, and T_2 denote the input, output, and transfer function of the successor machine. The transfer function T_3 of the serial connection is given by

$$T_3 = \frac{z_2}{x_1}.$$

However, since x_2 and z_1 are identical we have

$$T_3 = \frac{z_1}{x_1} \cdot \frac{z_2}{x_2} = T_1 \cdot T_2.$$

Similarly, the transfer function of a parallel connection of the above machines is given by $T_4 = T_1 + T_2$. The multiplication and addition of polynomials are performed over the $GF(p)$ field.

Example Let $T_1 = D^2 + 2D + 1$ and $T_2 = D + 1$ be transfer functions over the field $GF(3)$. The transfer functions, which correspond to the serial and parallel connections of T_1 and T_2, are given by

$$T_3 = (D^2 + 2D + 1)(D + 1) = D^3 + 1,$$
$$T_4 = (D^2 + 2D + 1) + (D + 1) = D^2 + 2.$$

Impulse response and null sequences

It is useful to define the *impulse response h* of an inert linear machine as its response to the input sequence $100\cdots0$. For example, the impulse response of the (inert) feedforward shift register of Fig. 15.4 is $a_0a_1a_2\cdots a_k0\cdots0$. After at most $k+1$ time units, the output of the k-dimensional feedforward shift register will be a sequence of 0's. In analogy to linear system theory, we can determine the response of an inert linear machine to an arbitrary input sequence from its impulse response. This is accomplished by performing a discrete "convolution" in $GF(p)$.

Example The impulse response of $T_1 = 1 + D + D^3$ is $h = 110100\cdots0$. The response of T_1 to the input sequence 1011 is obtained by addition (modulo 2) of the sequences h, D^2h, and D^3h, as follows:

Impulse:	1	0	0	0	0	·	·	0			
Impulse response h:	1	1	0	1	0	·	·	0			
Input sequence:	1	0	1	1							
h:	1	1	0	1	0	0	0	0	·	·	0
D^2h:	0	0	1	1	0	1	0	0	·	·	0
D^3h:	0	0	0	1	1	0	1	0	·	·	0
Output sequence:	1	1	1	1	1	1	1	0	·	·	0

The reader can similarly verify that the response of T_1 to the input sequence 11101 is 10000001.

If the initial state at $t = 0$ of an inert linear machine is $00\cdots0$, i.e., $y_1(0) = y_2(0) = \cdots = y_k(0) = 0$, and the input to the machine is a sequence of 0's then the output is also a sequence of 0's. However, it is possible to generate an output sequence consisting of 0's by providing the machine with a nonzero input sequence. Such a sequence is called a *null sequence* of the linear machine T and is denoted X_0, so that TX_0 is a sequence of 0's. If X_0 and X_0^* are null sequences for a machine T, that is, $TX_0 = 00\cdots0$ and $TX_0^* = 00\cdots0$, then $v_1TX_0 + v_2TX_0^* = T(v_1X_0 + v_2X_0^*) = 00\cdots0$, where v_1 and v_2 are scalars from $GF(p)$. Thus, *any linear combination of null sequences is also a null sequence for the machine.*

Example A null sequence of $T_1 = 1 + D + D^3$ is determined as follows:

$$0 = X_0 + DX_0 + D^3X_0,$$
$$X_0 = DX_0 + D^3X_0.$$

Thus, the present digit of X_0 is found by adding (modulo 2) the first and third past input digits of X_0. The null sequence is obtained by selecting an arbitrary nonzero sequence of length 3 (in general, of length equal to

dimension k) and specifying the subsequent digits. For T_1, the selection of 001 as the initial sequence yields the following null sequence:

$$X_0 = (0 \quad 0 \quad 1) \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1.$$

After seven digits the null sequence, which consists of the last seven digits, repeats itself.

Example The null sequence for the polynomial $T = 1 + 2D^2 + D^3$ over $GF(3)$ is found from

$$0 = X_0 + 2D^2X_0 + D^3X_0.$$

Adding $2X_0$ to both sides and recalling that $2X_0 + X_0 = 0$ in modulo 3 yields

$$2X_0 = 2D^2X_0 + D^3X_0.$$

Multiplying both sides by 2 yields

$$X_0 = D^2X_0 + 2D^3X_0.$$

Starting with 111, we obtain the null sequence

$$X_0 = (1\ 1\ 1)\ 0\ 0\ 2\ 0\ 2\ 1\ 2\ 2\ 1\ 0\ 2\ 2\ 2\ 0\ 0\ 1\ 0\ 1\ 2\ 1\ 1\ 2\ 0\ 1\ 1\ 1.$$

The preceding null sequences are known as *maximal* sequences, since each contains $(p^k - 1)$ digits and includes all possible k-tuples except $00\cdots0$. Additional properties of null sequences and their relationships to delay polynomials are discussed in [7].

Inverse machines

Feedforward shift registers are often used for encoding purposes. It is useful to determine whether an inverse machine that can be used as a decoder exists and, if it does, how to construct it. We shall say that a polynomial $T(D)$, where $z = Tx$, has an inverse, which will be denoted by $1/T(D)$, if there exists a network that realizes $x = (1/T)z$. We shall consider only those inverses that decode without any delay. The inverse of the feedforward shift register of Fig. 15.4 is obtained by reversing the directions of z and x in this schematic diagram and inverting the scalar multipliers, as shown in Fig. 15.6.

If we provide the inverse machine of Fig. 15.6 with the impulse response of the original machine of Fig. 15.4, i.e., $a_0a_1\cdots a_{k-1}a_k00\cdots0$, its response will be the original message, $x = 100\cdots0$. Since the inverse machine is linear and initially inert, it will decode any message produced by the original machine. (Note that negative scalars are actually positive integers since $(-a)$ modulo $p = (p - a)$ modulo p.)

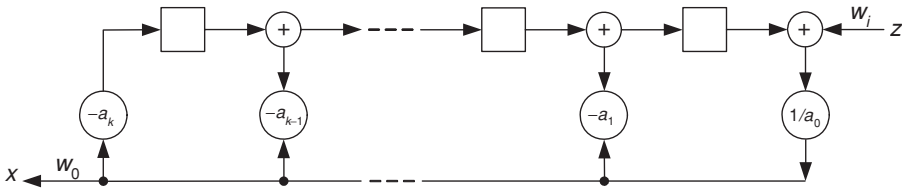


Fig. 15.6 Inverse machine for the shift register of Fig. 15.4.

From Fig. 15.6 it is evident that the inverse is realizable only if $a_0 \neq 0$. In general, *an inert linear machine described by a delay polynomial T has a linear inverse described by* T^{-1}*, which decodes without a delay, if and only if T contains a nonzero constant term that is prime to modulo p.* The general proof of this result is left to the reader as an exercise. The following demonstrates it for the case $GF(2)$.

The assertion is that an inert linear machine over the field of integers modulo 2 has an inverse, which decodes the output of the original machine without a delay, if and only if $a_0 = 1$ in T. To prove this assertion, consider the polynomial $T = a_1 D + a_2 D^2 + \cdots + a_k D^k$, for which $a_0 = 0$. Let the input to and the output from the inverse machine be denoted w_i and w_o, respectively; then the transfer function is given by

$$\frac{w_\mathrm{o}}{w_\mathrm{i}} = \frac{1}{a_1 D + a_2 D^2 + \cdots + a_k D^k}$$

or

$$a_1 D w_\mathrm{o} = w_i + a_2 D^2 w_\mathrm{o} + \cdots + a_k D^k w_\mathrm{o}.$$

The above equation means that a *past* output of the inverse machine (i.e., $D w_\mathrm{o}$) is a function of past outputs as well as the *present* input to the inverse machine. Such a condition is clearly not physically realizable. (If $a_1 = 0$, the above argument holds for the term containing the lowest order $a_i \neq 0$.)

If T does not contain a nonzero constant term, no instantaneous inverse can be found. However, an "inverse" that decodes the original input after a finite delay can be found. Let a_i be the scalar associated with the lowest power of D for which $a_i \neq 0$, i.e.,

$$T = D^i + a_{i+1} D^{i+1} + \cdots + a_k D^k$$

(modulo 2). The "inverse" is given by

$$\frac{w_\mathrm{o}}{w_\mathrm{i}} = \frac{1}{D^i + a_{i+1} D^{i+1} + \cdots + a_k D^k} \tag{15.4}$$

or

$$\frac{D^i w_\mathrm{o}}{w_\mathrm{i}} = \frac{1}{1 + a_{i+1} D + \cdots + a_k D^{k-i}}. \tag{15.5}$$

Although an inverse that decodes instantaneously does not exist for T, Eq. (15.5) corresponds to a realizable inverse, which regenerates the original message after a delay of i time units. Hence, if a sufficient finite delay is allowed then the messages generated by a feedforward shift register can always be decoded. This means that the shift register of Fig. 15.4 is actually lossless of order μ, where $\mu < k$.

Example The inverse of the inert linear machine of Fig. 15.5 is given by $T_1^{-1} = 1/(1 + D + D^3)$ and is shown in Fig. 15.7. (Note that for binary inert linear machines $-a_i = a_i$.)

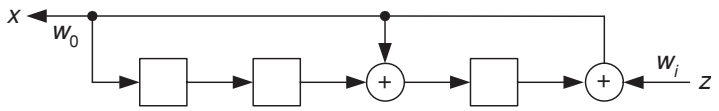
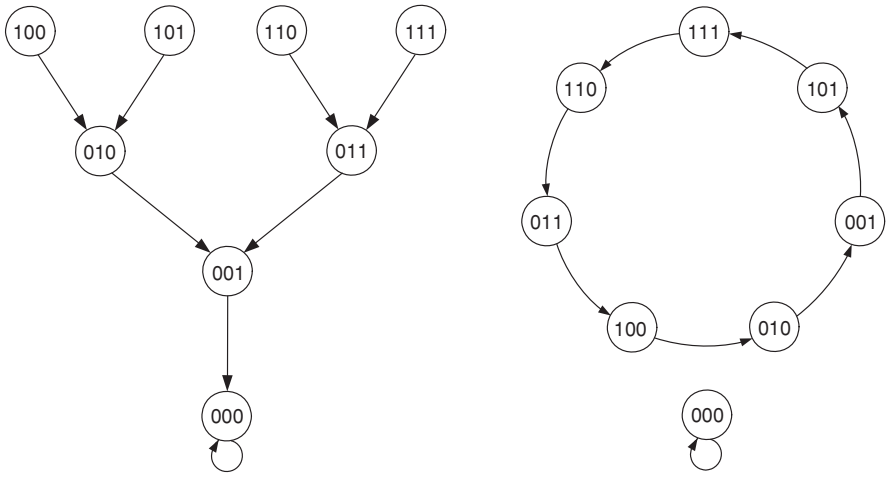


Fig. 15.7 The inverse of the machine in Fig. 15.5.

Linear machines with nonzero initial conditions

The inverse of an inert linear machine might not be inert. Consequently, its response to a sequence of zero input digits is not necessarily a sequence of zero output digits but could be a null sequence X_0 whose starting digits are determined by the initial state of the inverse. This can be shown by observing that the transfer function of the inverse is $x/z = T^{-1}$, or $z = Tx = 0$, because the input z to the inverse is assumed to be an all-0's sequence. Clearly, the solution of equation $Tx = 0$ is the null sequence X_0.

Let the input digits to the linear machine realizing T_1 and its inverse T_1^{-1} (Figs. 15.5 and 15.7, respectively) be 0's. If the machines are inert then their respective output digits will also be 0's. If, however, they are not inert then their respective output digits will not be 0's but will depend on their initial states. Since T_1 contains only feedforward paths, its response to a sequence of 0's might initially be nonzero, depending on the initial state. However, after at most three time units the response will be a sequence of 0's. In general, for every k-dimensional feedforward shift register the response to a sequence of 0's will also be a sequence of 0's, after a transient period of at most k time units in which the output digit might be nonzero. In the case of a noninert shift register that contains feedback paths, e.g., T_1^{-1}, the response to a sequence of 0's is not necessarily a sequence of 0's. The behavior of a noninert linear machine whose input is a sequence of 0's is often referred to as *autonomous* behavior, and it can be described by the state diagram of the corresponding machine whose input terminals are ignored. The state diagrams describing the autonomous behavior of the machines realizing T_1 and T_1^{-1} are given in Fig. 15.8.



(*a*) $T_1 = 1 + D + D^3$. (*b*) $T_1^{-1} = 1/(1 + D + D^3)$.

Fig. 15.8 State diagrams for autonomous behavior of linear machines.

An *autonomous linear machine* is a linear machine that contains no inputs (except a clock). A transition is caused by the clock pulse and, since the machine is deterministic, only one transition is permitted from each state. While the state diagram of T_1 contains only a single loop, corresponding to the case where the initial condition is 000, the diagram of T_1^{-1} contains two loops, which are called *cycle sets*. The nontrivial cycle in T_1^{-1} contains seven states and is maximal. (In general, the maximum number of distinct states in a k-dimensional modulo-p machine is p^k and, therefore, a maximal cycle contains $p^k - 1$ states.) For a more comprehensive study of the properties of autonomous linear machines, the reader is referred to Gill [9].

15.3 Inert linear machines and rational transfer functions

In the preceding section, the output of an inert linear machine was assumed to be a function of the present and some of the past input digits. In this section, we develop the more general case where the present output digit depends on the present and selected past input digits and also on a finite number of past output digits. In this latter case, the transfer function is a rational polynomial in the delay operator, i.e., $T = P(D)/Q(D)$.

Realization of rational polynomials

As an example, consider the inert linear machine whose output z is the modulo-2 sum of the present, first, second, and fourth previous input digits and of the first and third previous output digits, i.e.,

$$z = x + Dx + D^2x + D^4x + Dz + D^3z. \tag{15.6}$$

Equation (15.6) can be rewritten as

$$z(1 + D + D^3) = x(1 + D + D^2 + D^4)$$

and the transfer function is given by

$$T_2 = \frac{z}{x} = \frac{1 + D + D^2 + D^4}{1 + D + D^3}.$$

It can be shown that the numerator and denominator of T_2 do not contain any common factor and, thus, T_2 cannot be further simplified.

There are several methods for realizing the above transfer function. An obvious approach, although a very inefficient one, is to synthesize the inert linear machines given by the polynomials $1 + D + D^2 + D^4$ and $1/(1 + D + D^3)$ and to form a serial connection of these machines. Such a realization requires seven delay elements, four for the numerator and three for the denominator. Other synthesis procedures, which involve factoring of the numerator, partial fraction expansion, and ladder-type expansions, although useful do not necessarily yield a minimal realization. (A *minimal realization* is one that yields a machine of smallest dimension.) Clearly, the minimal possible dimension is determined by the degree of the polynomial and is equal to the highest degree in either the numerator or denominator of the transfer function. The chain realization described below yields a minimal realization in an efficient manner.

For T_2, the number of delay elements required in the minimal realization is four, the degree of the numerator. To demonstrate this assertion, let us rewrite Eq. (15.6) in increasing powers of D as follows:

$$x + z = D(x + z) + D^2x + D^3z + D^4x$$

or

$$x + z = D\{(x + z) + D[x + D(z + Dx)]\}. \tag{15.7}$$

The realization of Eq. (15.7), which is known as a *chain realization*, and that of its inverse, which corresponds to

$$T_2^{-1} = \frac{x}{z} = \frac{(1 + D + D^3)}{(1 + D + D^2 + D^4)},$$

are shown in Fig. 15.9. The output z is generated by adding x to $x + z$, which gives $(x + z) + x = z$ (modulo 2). This realization uses only EXCLUSIVE-OR adders, i.e., two-input modulo-2 adders, which are relatively inexpensive. In general, one characteristic of the chain realization is that it employs modulo-2 adders with only two inputs.

To obtain the chain realization of an arbitrary transfer function over $GF(2)$, note that the transfer function $T = P(D)/Q(D)$ of any realizable inert linear machine over $GF(p)$ has the form

$$T = \frac{z}{x} = \frac{a_0 + a_1D + \cdots + a_kD^k}{1 + b_1D + \cdots + b_kD^k} = \frac{P(D)}{Q(D)}, \tag{15.8}$$

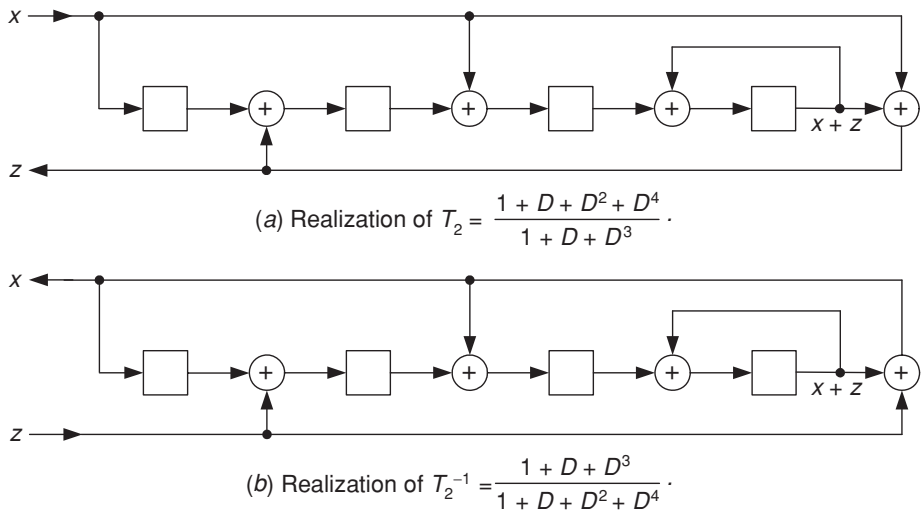


Fig. 15.9 Chain realization of an inert linear machine and its inverse.

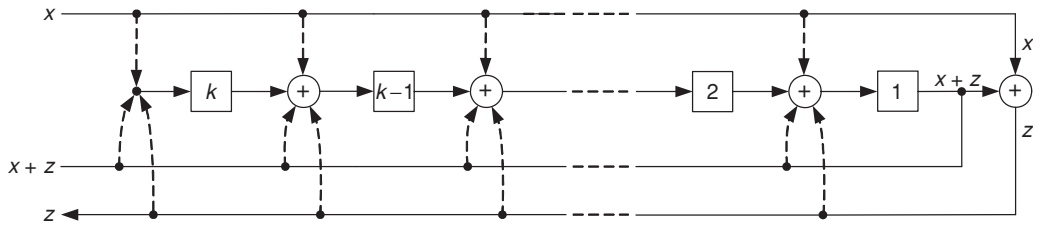


Fig. 15.10 Chain realization of an arbitrary transfer function over $GF(2)$.

where the a_i's and b_i's are elements of $GF(p)$. The denominator $Q(D)$ must contain the term 1 if T is to be realizable, as shown in the preceding section. Clearly, a realizable instantaneous inverse T^{-1} exists if and only if the numerator contains a nonzero constant term a_0 that is prime to modulo p. The machine T_2 has such an instantaneous inverse, as illustrated in Fig. 15.9*b*, since the numerator of T_2 contains a nonzero constant term, i.e., $a_0 = 1$.

For any invertible transfer function over $GF(2)$ of the form Eq. (15.8), we can write an expression for $x + z$ as a sum of *past* input and output digits, e.g., Eq. (15.7). This expression can be realized by an alternating chain of delay elements and modulo-2 adders, as shown in Fig. 15.10. In general, the chain realization of a k-dimensional inert linear machine requires k delay elements and at most k two-input modulo-2 adders. One input to the ith adder from the right (except the first adder) is the output of the ith delay element. The second input, if required, is x, z, or $x + z$, depending respectively on whether the term D^{i-1} is present in the numerator or denominator of T or both. The second input to the rightmost adder is always x, so that $x + (x + z) = z$. If D^{i-1} is absent from both $P(D)$ and $Q(D)$, i.e., $a_{i-1} = b_{i-1} = 0$, no second input is required and the ith adder may be deleted. The inverse machine

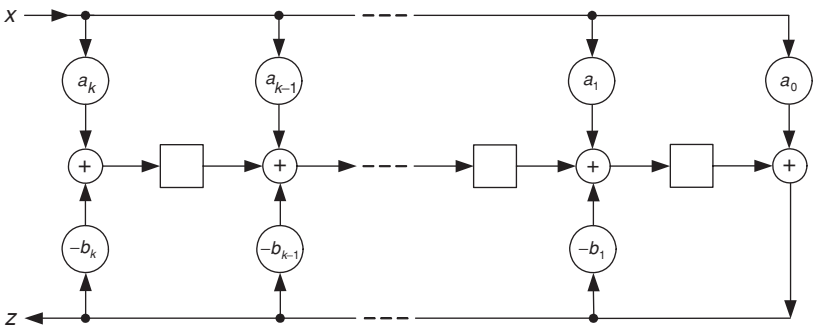


Fig. 15.11 Realization of $T = (a_0 + a_1 D + \cdots + a_k D^k)/(1 + b_1 D + \cdots + b_k D^k)$(modulo p).

is obtained simply by interchanging the roles of x and z, as illustrated in Fig. 15.9*b*.

The realization of a two-terminal k-dimensional inert linear machine, over the $GF(p)$ field, whose transfer function is given by Eq. (15.8), is shown in Fig. 15.11. Note that for $p \geq 3$ it is generally not sufficient to employ only two-terminal adders, unless the number of adders is increased. The realization of Fig. 15.11 is obtained in a direct manner from the realizations in Figs. 15.4 and 15.6. The verification that it indeed realizes Eq. (15.8) is left to the reader as an exercise.

Example The realization of

$$T_3 = \frac{1 + 2D + D^2 + 2D^4}{1 + 2D + D^2 + D^3}$$

over $GF(3)$ is shown in Fig. 15.12.

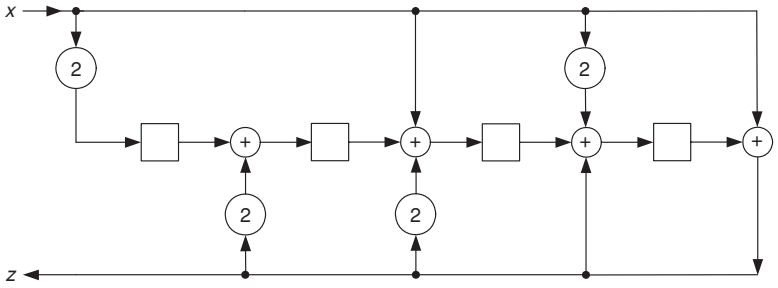


Fig. 15.12 Realization of $T_3 = (1 + 2D + D^2 + 2D^4)/(1 + 2D + D^2 + D^3)$(modulo 3).

Impulse response and transfer function

The impulse response h of an inert linear machine has been defined as its response to the input sequence $100 \cdots 0$. For any given impulse response, a transfer function can always be specified and if the impulse response is realizable then a corresponding machine can be synthesized. We shall now

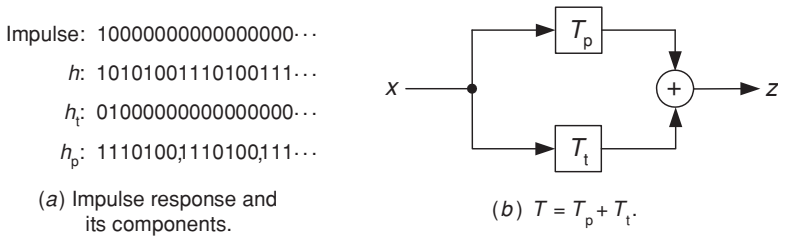


Fig. 15.13 Synthesis of an inert linear machine from its impulse response.

show how to synthesize an inert linear machine from its impulse response. In particular, we shall prove that if the impulse response is realizable then it consists of two components: a transient component denoted h_t and a periodic component denoted h_p.

In Section 10.2, it was shown that the response of an arbitrary sequential machine to a periodic excitation is periodic. In particular, the response to a sequence of 0's is periodic with period shorter than or equal to n, where n is the number of states. For a k-dimensional inert linear machine, the period of the response to a sequence of 0's is at most $p^k - 1 = n - 1$, since this is the maximal nontrivial cycle set (excluding the zero state). Consequently, a necessary condition for an impulse response h to be realizable is that it will ultimately become periodic. In addition, since the length of the transient response is at most $k + 1$, the transfer function of a realizable two-terminal k-dimensional inert linear machine can be specified uniquely by observing the first $k + p^k$ symbols of the impulse response:

As an example, consider the impulse response $h = 1010100$, $1110100, 1110100, \ldots$ of an inert linear machine over $GF(2)$. The impulse response can be separated into a transient and a periodic component such that $h = h_t + h_p$, as shown in Fig. 15.13*a*. The synthesis of the corresponding inert linear machine can be accomplished by specifying separately transfer functions T_t and T_p, corresponding, respectively, to h_t and h_p, such that the overall transfer function $T = T_t + T_p$ (see Fig. 15.13*b*). The transfer function T_t is found from h_t to equal D. The periodic component h_p can be described by $1 + D + D^2 + D^4$ and, since the period is 7, the entire periodic transfer function is specified by

$$T_p = (1 + D + D^2 + D^4)(1 + D^7 + D^{14} + \cdots)$$

or

$$T_p = \frac{1 + D + D^2 + D^4}{1 + D^7}.$$

Hence,

$$\begin{aligned} T = T_p + T_t &= \frac{1 + D + D^2 + D^4}{1 + D^7} + D \\ &= \frac{1 + D^2 + D^4 + D^8}{1 + D^7}. \end{aligned}$$

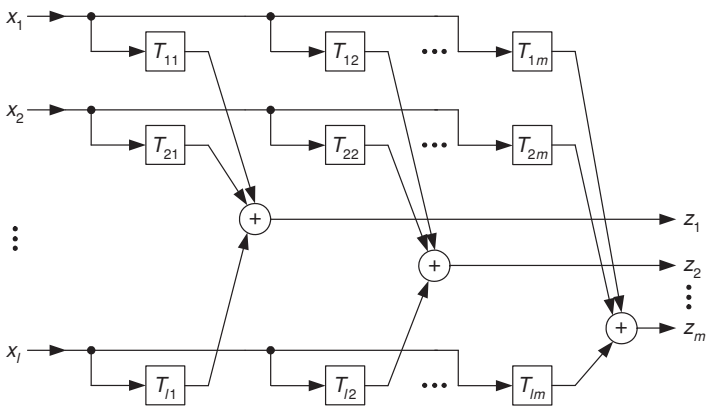


Fig. 15.14 Schematic diagram of a multi-terminal inert linear machine.

This function can be simplified as (see Appendix 15.2)

$$T = \frac{(1 + D + D^2 + D^4)^2}{(1 + D + D^2 + D^4)(1 + D + D^3)} = \frac{1 + D + D^2 + D^4}{1 + D + D^3}.$$

A minimal realization of this transfer function is shown in Fig. 15.9*a*.

Multi-terminal machines

In the preceding sections we developed the properties of two-terminal inert linear machines characterized by rational polynomials in the delay operator D. A multi-terminal inert linear machine with l input terminals and m output terminals can be characterized by a set of lm transfer functions, where

$$T_{ij}(D) = \frac{z_j}{x_i} \quad \text{for all } i = 1, 2, \ldots, l \quad \text{and} \quad j = 1, 2, \ldots, m.$$

The transfer function T_{ij} is evaluated when $x_i = 0$ for all $i \neq j$; i.e., T_{ij} specifies the dependency of output z_j on input x_i when all other inputs are held at zero. The synthesis problem of a multi-terminal inert linear machine can thus be transformed to the well-known problem of synthesizing a set of two-terminal inert linear machines. A realization of an arbitrary multi-terminal inert linear machine from an appropriate set of two-terminal machines is shown in Fig. 15.14. It must be emphasized that this is not always a minimal realization; rather, it demonstrates that a realization exists. More efficient methods, that yield minimal realizations, are developed in subsequent sections.

15.4 The general model

The specification of the outputs z_j of an inert linear machine by means of a set of polynomials, such that $z_j = \sum_{i=1}^{l} T_{ij} x_i$, is actually a "black box" type

of specification; that is, each output is specified in terms of only the external inputs and the characterizing polynomials. Such a specification is possible since the machine is assumed to be initially inert, i.e., $x(t) = 0$ for all $t < 0$ and, therefore, $y_i(t) = 0$ for all $t < 0$ and $i = 1, 2, \ldots, k$. The specification of an arbitrary (not necessarily inert) linear machine is accomplished by specifying the output and next-state functions in terms of the inputs as well as the present states of the machine.

The matrix formulation

Consider a k-dimensional linear machine over $GF(p)$, with l inputs and m outputs, as shown in Fig. 15.1. Since the combinational logic consists of only adders and scalar multipliers, the next state of the delay Y_i can be expressed as a function of the external inputs of the machine and its present state, as follows:

$$Y_i = (\alpha_{i1}y_1 + \alpha_{i2}y_2 + \cdots + \alpha_{ik}y_k) + (\beta_{i1}x_1 + \beta_{i2}x_2 + \cdots + \beta_{il}x_l)$$

or

$$Y_i = \sum_{j=1}^{k} \alpha_{ij}y_j + \sum_{j=1}^{l} \beta_{ij}x_j. \tag{15.9}$$

Equation (15.9) is called the *next-state equation* for delay Y_i. The entire set of next-state equations for a given machine can be expressed compactly in a matrix form as follows:

$$\begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_k \end{bmatrix} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1k} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2k} \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_{k1} & \alpha_{k2} & \cdots & \alpha_{kk} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix} + \begin{bmatrix} \beta_{11} & \beta_{12} & \cdots & \beta_{1l} \\ \beta_{21} & \beta_{22} & \cdots & \beta_{2l} \\ \vdots & \vdots & \vdots & \vdots \\ \beta_{k1} & \beta_{k2} & \cdots & \beta_{kl} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_l \end{bmatrix} \tag{15.10}$$

or

$$\mathbf{Y}(t) = \mathbf{y}(t+1) = \mathbf{A}\mathbf{y}(t) + \mathbf{B}\mathbf{x}(t).$$

The vector $\mathbf{y}(t)$ is called the *present-state vector*; its elements are the state variables. The vector $\mathbf{Y}(t)$ is the *next-state vector*, where $\mathbf{Y}(t) = \mathbf{y}(t+1)$. The vector $\mathbf{x}(t)$ is the *input vector*; its elements are the *input variables*, where $x_i(t)$ is the input applied to the ith terminal at time t. The dimensions of the state and input vectors are k and l, respectively, i.e.,

$$\mathbf{y}(t) = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix}, \qquad \mathbf{Y}(t) = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_k \end{bmatrix}, \qquad \mathbf{x}(t) = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_l \end{bmatrix}.$$

When the dependence on t is understood, $y_i(t)$ and $x_i(t)$ are written as y_i and x_i, respectively.

In a similar manner, each output function can be specified in terms of the present state and inputs of the machine. The ith output is expressed as

$$z_i = (\gamma_{i1}y_1 + \gamma_{i2}y_2 + \cdots + \gamma_{ik}y_k) + (\delta_{i1}x_1 + \delta_{i2}x_2 + \cdots + \delta_{il}x_l)$$

or

$$z_i = \sum_{j=1}^{k} \gamma_{ij}y_j + \sum_{j=1}^{l} \delta_{ij}x_j. \tag{15.11}$$

Equation (15.11) is called the *output equation*. The entire set of output equations for a given machine can also be expressed in a matrix form, as follows:

$$\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \cdots & \gamma_{1k} \\ \gamma_{21} & \gamma_{22} & \cdots & \gamma_{2k} \\ \vdots & \vdots & \vdots & \vdots \\ \gamma_{m1} & \gamma_{m2} & \cdots & \gamma_{mk} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix} + \begin{bmatrix} \delta_{11} & \delta_{12} & \cdots & \delta_{1l} \\ \delta_{21} & \delta_{22} & \cdots & \delta_{2l} \\ \vdots & \vdots & \vdots & \vdots \\ \delta_{m1} & \delta_{m2} & \cdots & \delta_{ml} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_l \end{bmatrix} \tag{15.12}$$

or

$$\mathbf{z}(t) = \mathbf{C}\mathbf{y}(t) + \mathbf{D}\mathbf{x}(t),$$

where $\mathbf{z}(t)$ is the *output vector*; its ith element $z_i(t)$ is the output generated at terminal i at time t.

The matrices **A**, **B**, **C**, and **D** defined by Eqs. (15.10) and (15.12) are the *characterizing matrices* of the linear machine; **A** is referred to as the *characteristic matrix* and specifies the autonomous behavior of the machine. The matrix formulation completely characterizes any linear machine, and thus it leads to a precise definition of a linear machine in terms of the characterizing matrices, as follows.

Definition 15.1 A machine is said to be *linear* over a finite field $GF(p)$ if its states can be identified with the elements of a vector space and its next-state and output functions can be specified by a pair of matrix equations over $GF(p)$,

$$\mathbf{Y}(t) = \mathbf{A}\mathbf{y}(t) + \mathbf{B}\mathbf{x}(t), \tag{15.13}$$

$$\mathbf{z}(t) = \mathbf{C}\mathbf{y}(t) + \mathbf{D}\mathbf{x}(t). \tag{15.14}$$

The *dimension* of the machine is the dimension of its state vector.

Equations (15.13) and (15.14) represent a Moore or Mealy machine, according to whether **D** is or is not identically zero. We subsequently refer to a machine whose characterizing matrices are **A**, **B**, **C**, and **D** as the machine $\{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$.

The elements of the characterizing matrices are determined from the next-state and output equations, Eqs. (15.9) and (15.11) respectively, in the following manner. The coefficient α_{ij} denotes the product of scalar multipliers contained

in the path leading from y_j to Y_i. If there are two or more paths from y_j to Y_i, α_{ij} denotes the sum of all such products; if no path exists between y_j and Y_i, $\alpha_{ij} = 0$. The coefficient β_{ij} denotes the corresponding values for the paths leading from the input x_j to Y_i. Similarly, γ_{ij} denotes the sum of products of the scalar multipliers contained in the paths leading from y_j to output terminal z_i; if no path exists between y_j and z_i then $\gamma_{ij} = 0$. The coefficient δ_{ij} denotes the corresponding values for paths originating at input x_j and terminating at output z_i.

Example The characterizing matrices for the four-terminal linear machine of Fig. 15.3 are

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 2 & 2 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 2 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 0 & 2 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} 0 & 2 \\ 0 & 1 \end{bmatrix}.$$

The response of linear machines

The relationship between the input sequence to the machine $\{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$ and its corresponding output sequence is obtained by iterating Eqs. (15.13) and (15.14), i.e.,

$$\begin{aligned}
\mathbf{y}(1) &= \mathbf{A}\mathbf{y}(0) + \mathbf{B}\mathbf{x}(0), \\
\mathbf{z}(0) &= \mathbf{C}\mathbf{y}(0) + \mathbf{D}\mathbf{x}(0), \\
\mathbf{z}(1) &= \mathbf{C}\mathbf{A}\mathbf{y}(0) + \mathbf{C}\mathbf{B}\mathbf{x}(0) + \mathbf{D}\mathbf{x}(1), \\
\mathbf{z}(2) &= \mathbf{C}\mathbf{A}^2\mathbf{y}(0) + \mathbf{C}\mathbf{A}\mathbf{B}\mathbf{x}(0) + \mathbf{C}\mathbf{B}\mathbf{x}(1) + \mathbf{D}\mathbf{x}(2), \\
\mathbf{z}(3) &= \mathbf{C}\mathbf{A}^3\mathbf{y}(0) + \mathbf{C}\mathbf{A}^2\mathbf{B}\mathbf{x}(0) + \mathbf{C}\mathbf{A}\mathbf{B}\mathbf{x}(1) + \mathbf{C}\mathbf{B}\mathbf{x}(2) + \mathbf{D}\mathbf{x}(3), \\
&\vdots \\
\mathbf{z}(t) &= \mathbf{C}\mathbf{A}^t\mathbf{y}(0) + \sum_{j=0}^{t-1} \mathbf{C}\mathbf{A}^{t-1-j}\mathbf{B}\mathbf{x}(j) + \mathbf{D}\mathbf{x}(t)
\end{aligned}$$

or

$$\mathbf{z}(t) = \mathbf{C}\mathbf{A}^t\mathbf{y}(0) + \sum_{j=0}^{t} \mathbf{H}(t-j)\mathbf{x}(j) \tag{15.15}$$

where

$$\mathbf{H}(t-j) = \begin{cases} \mathbf{D} & \text{when } t - j = 0, \\ \mathbf{C}\mathbf{A}^{t-1-j}\mathbf{B} & \text{when } t - 1 - j \geq 0. \end{cases} \tag{15.16}$$

From Eq. (15.15) we see that the response of a linear machine consists of two components. The first component, known as the *autonomous response*, is obtained by setting $\mathbf{x}(t) = \mathbf{0}$ for all $t \geq 0$, i.e.,

$$\mathbf{z}_{\mathrm{a}}(t) = \mathbf{C}\mathbf{A}^t\mathbf{y}(0). \tag{15.17}$$

The second component, known as the *forced response*, is obtained by setting $\mathbf{y}(0) = \mathbf{0}$, i.e.,

$$\mathbf{z}_{\mathrm{f}}(t) = \sum_{j=0}^{t} \mathbf{H}(t-j)\mathbf{x}(j). \tag{15.18}$$

The total response is thus given by

$$\mathbf{z}(t) = \mathbf{z}_{\mathrm{a}}(t) + \mathbf{z}_{\mathrm{f}}(t). \tag{15.19}$$

Equation (15.18) actually describes the response of inert machines in matrix form. These machines have been studied extensively in earlier sections by means of the polynomial representation. The total response, Eq. (15.19), of a linear machine for a given input sequence and an arbitrary initial state can be found by separately determining the forced and autonomous responses and adding them up.

The autonomous response is generally determined from the analysis of the internal circuit.[3] The state behavior of the internal circuit is completely characterized by the characteristic matrix $\mathbf{A}$, since Eq. (15.13) becomes

$$\mathbf{Y}(t) = \mathbf{y}(t+1) = \mathbf{A}\mathbf{y}(t).$$

Because the internal circuit is autonomous, the λ-successor S_j of state S_i, where $S_i = \mathbf{y}_i(t)$, is given by

$$\mathbf{y}_j(t) = \mathbf{A}^{\lambda}\mathbf{y}_i(t)$$

where λ denotes the number of state transitions. (Note that while y_j denotes the state of the jth delay, $\mathbf{y}_i$ denotes the state S_i of the machine.)

The sequence of predecessors of a given state is established by constructing the inverse internal circuit; such an inverse exists only if each state has a unique predecessor. For an internal circuit given by $\mathbf{A}$, the inverse is given by $\mathbf{A}^{-1}$ since

$$\mathbf{y}(t) = \mathbf{A}^{-1}\mathbf{Y}(t).$$

Thus, the inverse circuit exists if and only if $\mathbf{A}$ is nonsingular, i.e., the determinant $|\mathbf{A}|$ is nonzero.

Autonomous linear machines are best analyzed either by means of their state diagrams (as illustrated earlier in Fig. 15.8) or by means of the characteristic polynomials derived from $\mathbf{A}$. For further discussion on autonomous linear machines, see [9].

15.5 Reduction of linear machines

We now determine conditions, in terms of characterizing matrices, for linear machines to be finite-memory and definitely diagnosable. The length of

[3] The *internal circuit* is that part of the circuit that can be specified by $\mathbf{A}$ alone, that is, it contains only the delay elements and their interconnections; the input and output lines have been deleted.

the shortest distinguishing sequence for arbitrary initial uncertainty will be obtained. A procedure will be presented to determine whether a given linear machine is minimal and, if it is not, how to minimize it. The techniques developed in earlier chapters for arbitrary sequential machines are valid for linear machines as well. Our current objective, however, is to develop an *analytical* procedure, rather than an enumerative one, which is valid only for linear machines and which utilizes the matrix formulation.

The diagnostic matrix

Let L be a k-dimensional linear machine over $GF(p)$. To describe an experiment of length k, Eqs. (15.15) and (15.16) can be expressed compactly as

$$\mathbf{Z}^{(k)} = \mathbf{K}_k \mathbf{y}(0) + \mathbf{V}_k \mathbf{X}^{(k)}, \tag{15.20}$$

where

$$\mathbf{Z}^{(k)} = \begin{bmatrix} \mathbf{z}(0) \\ \mathbf{z}(1) \\ \vdots \\ \mathbf{z}(k-1) \end{bmatrix}, \quad \mathbf{K}_k = \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \\ \vdots \\ \mathbf{CA}^{k-1} \end{bmatrix}, \quad \mathbf{X}^{(k)} = \begin{bmatrix} \mathbf{x}(0) \\ \mathbf{x}(1) \\ \vdots \\ \mathbf{x}(k-1) \end{bmatrix},$$

and

$$\mathbf{V}_k = \begin{bmatrix} \mathbf{D} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{CB} & \mathbf{D} & \mathbf{0} & \cdots & \cdot \\ \mathbf{CAB} & \mathbf{CB} & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \mathbf{0} \\ \mathbf{CA}^{k-2}\mathbf{B} & \cdot & \cdot & \cdots & \mathbf{D} \end{bmatrix}.$$

The vector $\mathbf{y}(0)$ denotes the initial state at $t = 0$. For initial states S_a and S_b, the corresponding state vectors are denoted $\mathbf{y}_a(0)$ and $\mathbf{y}_b(0)$, respectively. The matrix $\mathbf{K}_k$, which consists of submatrices corresponding to the different outputs, is called the *diagnostic* (or *distinguishing*) matrix.

From Eq. (15.20) it is evident that if S_a is equivalent to S_b then

$$\mathbf{K}_k \mathbf{y}_a(0) = \mathbf{K}_k \mathbf{y}_b(0), \tag{15.21}$$

since the second term $\mathbf{V}_k \mathbf{X}^{(k)}$ is independent of the initial state and depends only on the input sequence. Moreover, since the inputs enter Eq. (15.20) additively, all input sequences are equally effective in state-distinguishing experiments. Consequently, to simplify the computation $\mathbf{X}^{(k)}$ may be selected as the all-zero sequence $\mathbf{X}^{(k)} = \mathbf{0}$, reducing Eq. (15.20) to

$$\mathbf{Z}^{(k)} = \mathbf{K}_k \mathbf{y}(0). \tag{15.22}$$

The proof that Eq. (15.21) is a necessary and sufficient condition for S_a and S_b to be equivalent follows from Theorem 15.1, and is left to the reader as an exercise.

Before proceeding with the investigation of the minimal linear machines, it is necessary to show that the first r linearly independent rows of the diagnostic matrix $\mathbf{K}_k$ occur in a consecutive sequence in $\mathbf{C}, \mathbf{CA}, \ldots, \mathbf{CA}^i$, where $i < r$. To prove this assertion, assume that all the rows of $\mathbf{CA}^i$ are linear combinations of the rows of $\mathbf{K}_i$, i.e., the rows of $\mathbf{C}, \mathbf{CA}, \ldots, \mathbf{CA}^{i-1}$. Then the rows of $\mathbf{CA}^{i+1}$ are the same linear combinations of rows of $\mathbf{K}_i\mathbf{A}$, i.e., $\mathbf{CA}$, $\mathbf{CA}^2, \ldots, \mathbf{CA}^i$. However, since the rows of $\mathbf{CA}^i$ are linear combinations of the rows of $\mathbf{C}, \mathbf{CA}, \ldots, \mathbf{CA}^{i-1}$, the rows of $\mathbf{CA}^{i+1}$ are also linear combinations of the rows of $\mathbf{C}, \mathbf{CA}, \ldots, \mathbf{CA}^{i-1}$. Consequently, the process of finding the linearly independent rows of $\mathbf{K}_k$ terminates as soon as some submatrix $\mathbf{CA}^i$ is generated whose rows are linearly dependent on the rows of the preceding submatrices.

Theorem 15.1 *A k-dimensional linear machine* $\{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$ *is definitely diagnosable of order k if and only if diagnostic matrix* $\mathbf{K}_k$ *has k linearly independent rows.*

Proof The state vector $\mathbf{y}$ is k-dimensional and consequently $\mathbf{K}_k$ has exactly k columns. Thus, the rank of $\mathbf{K}_k$ cannot exceed k. If $\mathbf{K}_k$ contains k linearly independent rows then, under a sequence of all-zero inputs, the outputs corresponding to these rows in Eq. (15.22) impose k linearly independent constraints on $\mathbf{y}(0)$. Since $\mathbf{y}(0)$ is k-dimensional, it is specified uniquely by these constraints and thus the all-zero sequence of length k is a distinguishing sequence. However, since all input sequences of a given length have been shown to be equally effective in distinguishing experiments, every input sequence of length k or more is a distinguishing sequence and the machine is definitely diagnosable.

To prove that it is definitely diagnosable of order k, it is sufficient to note that the rows of $\mathbf{CA}^k, \mathbf{CA}^{k+1}, \ldots$ are linearly dependent on the rows of $\mathbf{K}_k$ and thus the length of distinguishing sequences need not exceed the rank of $\mathbf{K}_k$. If $\mathbf{K}_k$ contains fewer than k linearly independent rows, there must exist some nonzero $\mathbf{y}(0) \neq \mathbf{0}$ that is annihilated by $\mathbf{K}_k$ and, hence, results in the same input–output behavior as in the case $\mathbf{y}(0) = \mathbf{0}$. This means that the machine in question is not reduced. ◇

From Theorem 15.1, it follows that a linear machine is in reduced form if and only if the rank of $\mathbf{K}_k$ is k. Moreover, *every reduced k-dimensional linear machine is definitely diagnosable of order k and is finite-memory of order less than or equal to k.* These properties are also known as the *observability* and *predictability* properties of linear machines.

Example Consider the linear machine L_1 over $GF(2)$ given by the following matrices

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

The diagnostic matrix $\mathbf{K}_3$ is obtained:

$$\mathbf{K}_3 = \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \\ \mathbf{CA}^2 \end{bmatrix}.$$

Thus Eq. (15.22) becomes

$$\begin{bmatrix} z_1(0) \\ z_2(0) \\ z_1(1) \\ z_2(1) \\ z_1(2) \\ z_2(2) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} y_1(0) \\ y_2(0) \\ y_3(0) \end{bmatrix}.$$

The rank of $\mathbf{K}_3$ is 3 and hence the dimension of L_1 cannot be reduced. For a given initial state, the values of $y_1(0)$, $y_2(0)$, and $y_3(0)$ are specified, and the matrix $\mathbf{Z}^{(t)}$ yields the response of L_1 to the distinguishing sequence 000. For example, if the initial state is (111) then in response to 000 the sequences $z_1 = 010$ and $z_2 = 100$ are produced. It is suggested that the reader should draw the circuit diagram and compare actual circuit responses with responses obtained in an analytical manner.

The minimization procedure

Let L be a k-dimensional linear machine $\{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$ over $GF(p)$ and let r be the rank of the diagnostic matrix, where $r < k$. Define an $r \times k$ matrix $\mathbf{T}$ consisting of the first r linearly independent rows of $\mathbf{K}_k$, and a $k \times r$ matrix $\mathbf{R}$ denoting the right inverse of $\mathbf{T}$, such that $\mathbf{TR} = \mathbf{I}_r$ where $\mathbf{I}_r$ is the $r \times r$ identity matrix. Define an r-dimensional machine L^* with characterizing matrices $\{\mathbf{A}^*, \mathbf{B}^*, \mathbf{C}^*, \mathbf{D}^*\}$. such that

$$\mathbf{A}^* = \mathbf{TAR}, \quad \mathbf{B}^* = \mathbf{TB}, \quad \mathbf{C}^* = \mathbf{CR}, \quad \mathbf{D}^* = \mathbf{D}. \tag{15.23}$$

At this point, we shall state and prove a major theorem that establishes the validity of the following minimization procedure.

Theorem 15.2 *The State* $\mathbf{y}$ *of* L *is equivalent to the state* $\mathbf{y}^* = \mathbf{Ty}$ *of* L^*. *The machine* L^* *is a reduced machine equivalent to* L.

Proof[4] In order to prove the first part, it is necessary and sufficient to show that, for every state of L, $\mathbf{y}^*$ and $\mathbf{Ty}$ have equivalent successors and yield identical output digits, i.e.,

$$\mathbf{T}(\mathbf{Ay} + \mathbf{Bx}) = \mathbf{A}^*\mathbf{y}^* + \mathbf{B}^*\mathbf{x}$$

and

$$\mathbf{Cy} + \mathbf{Dx} = \mathbf{C}^*\mathbf{y}^* + \mathbf{D}^*\mathbf{x}.$$

Define $\bar{\mathbf{y}} = \mathbf{y} - \mathbf{RTy}$; then, since $\mathbf{TR} = \mathbf{I}_r$ we obtain

$$\mathbf{T}\bar{\mathbf{y}} = \mathbf{Ty} - \mathbf{TRTy} = \mathbf{Ty} - \mathbf{Ty} = \mathbf{0}.$$

Since $\mathbf{T}\bar{\mathbf{y}} = \mathbf{0}$, we have $\mathbf{K}_k\bar{\mathbf{y}} = \mathbf{0}$. Therefore, by Eq. (15.21), state $\bar{\mathbf{y}}$ is equivalent to state $\mathbf{0}$. In addition, since $\mathbf{A0} = \mathbf{0}$,

$$\mathbf{A}\bar{\mathbf{y}} = \mathbf{0} \qquad \text{and} \qquad \mathbf{TA}\bar{\mathbf{y}} = \mathbf{0}.$$

Also, since the rows of $\mathbf{C}$ are spanned by those of $\mathbf{T}$, $\mathbf{C}\bar{\mathbf{y}} = \mathbf{0}$. The next-state and output equations are

$$\begin{aligned}\mathbf{T}(\mathbf{Ay} + \mathbf{Bx}) &= \mathbf{T}[\mathbf{A}(\bar{\mathbf{y}} + \mathbf{RTy}) + \mathbf{Bx}] = \mathbf{TA}\bar{\mathbf{y}} + \mathbf{TARTy} + \mathbf{TBx} \\ &= \mathbf{0} + (\mathbf{TAR})(\mathbf{Ty}) + (\mathbf{TB})\mathbf{x} = \mathbf{A}^*\mathbf{y}^* + \mathbf{B}^*\mathbf{x}, \\ \mathbf{Cy} + \mathbf{Dx} &= \mathbf{C}(\bar{\mathbf{y}} + \mathbf{RTy}) + \mathbf{Dx} = \mathbf{C}\bar{\mathbf{y}} + \mathbf{CRTy} + \mathbf{Dx} \\ &= \mathbf{0} + (\mathbf{CR})(\mathbf{Ty}) + \mathbf{Dx} = \mathbf{C}^*\mathbf{y}^* + \mathbf{D}^*\mathbf{x}.\end{aligned}$$

Hence, $\mathbf{y}^* = \mathbf{Ty}$ under the transformation of Eq. (15.23). Similarly, since $\mathbf{Ry}^* = \mathbf{RTy} = \mathbf{y}$, the state $\mathbf{y}^*$ of L^* is equivalent to the state $\mathbf{y} = \mathbf{Ry}^*$ of L.

We shall now show that L^* is a reduced machine and thus is the minimal machine equivalent to L. Since $\mathbf{K}_k$ has rank less than k, it partitions the states of L into subsets (usually called cosets) as follows. Let G_0 denote the subset containing all states that are equivalent to the zero state $\mathbf{y} = \mathbf{0}$. From Eq. (15.21) we conclude that G_0 denotes the null space of $\mathbf{K}_k$. Let us now generate a set of subsets from G_0 such that two states $\mathbf{y}_a$ and $\mathbf{y}_b$ are in the same subset if and only if $\mathbf{y}_a - \mathbf{y}_b$ is in G_0. Hence, $\mathbf{K}_k(\mathbf{y}_a - \mathbf{y}_b) = \mathbf{0}$ and $\mathbf{K}_k\mathbf{y}_a = \mathbf{K}_k\mathbf{y}_b$, which means that $\mathbf{y}_a$ is equivalent to $\mathbf{y}_b$ and the subsets so generated are the equivalence classes of L. Moreover, since states in different subsets are distinguishable by the all-zero sequence (or any other input sequence), the subsets generated by $\mathbf{K}_k$ correspond to states of the reduced form of the original machine. (These subsets are actually identical to the blocks of the final partition in the reduction procedure outlined in Chapter 10.)

Since G_0 generates $p^r - 1$ distinct subsets, the reduced form of L over $GF(p)$ has p^r states, where r is the rank of $\mathbf{K}_k$. Since L and L^* are equivalent and L^* has exactly p^r states, it is the minimal machine equivalent to L. ◇

[4] This proof requires some knowledge of matrix algebra and may be skipped at first reading.

Example Consider the linear machine L_2 over $GF(2)$ defined by the matrices

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} 1 \end{bmatrix},$$

$$\mathbf{K}_3 = \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \\ \mathbf{CA}^2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

The rank of $\mathbf{K}_3$ is 2 and thus L_2 is reducible. The first two rows of $\mathbf{K}_3$ are linearly independent; therefore

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

The right inverse $\mathbf{R}$ of $\mathbf{T}$ is constructed by selecting a set of r linearly independent columns from $\mathbf{T}$. Since the rank of $\mathbf{T}$ is r and column rank equals row rank, such a set always exists. Form an $r \times r$ matrix $\mathbf{Q}$ from these columns and find its inverse, $\mathbf{Q}^{-1}$. The right inverse $\mathbf{R}$, which is a $k \times r$ matrix, is formed by placing in it the rows of $\mathbf{Q}^{-1}$ in positions corresponding to the columns selected from $\mathbf{T}$, all other rows being set to zero.

In our case,

$$\mathbf{Q} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{Q}^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{R} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}.$$

Following the definitions of the characterizing matrices of L_2^*, we obtain

$$\mathbf{y}^* = \mathbf{Ty} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \mathbf{y}$$

$$\mathbf{A}^* = \mathbf{TAR} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

$$\mathbf{B}^* = \mathbf{TB} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

$$\mathbf{C}^* = \mathbf{CR} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix},$$

$$\mathbf{D}^* = \mathbf{D} = [1]\cdot$$

The circuit diagram of the reduced machine L_2^* given by $\{\mathbf{A}^*, \mathbf{B}^*, \mathbf{C}^*, \mathbf{D}^*\}$ is shown in Fig. 15.15.

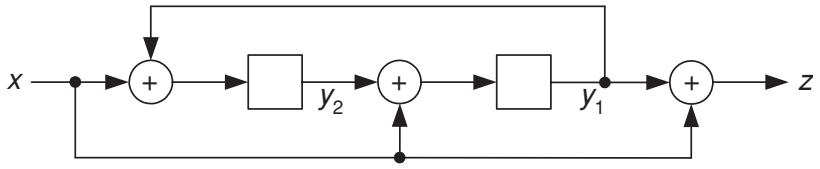


Fig. 15.15 Realization of the reduced machine L_2^*.

The minimal machine L_2^* has been obtained without explicitly constructing the equivalence classes of L_2. We shall now find them to demonstrate the procedure outlined in the proof of Theorem 15.2. From Eq. (15.22), we have

$$\begin{bmatrix} z_1(0) \\ z_1(1) \end{bmatrix} = \mathbf{Ty}(0) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} y_1(0) \\ y_2(0) \\ y_3(0) \end{bmatrix}. \tag{15.24}$$

Here G_0 contains all the states, designated by their corresponding vectors, for which $\mathbf{0} = \mathbf{Ty}(0)$, i.e.,

$$G_0 = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\}.$$

The remaining subsets, which yield equivalence classes of L_2, are obtained by adding to G_0 any element not contained in it and such that two states $\mathbf{y}_a$ and $\mathbf{y}_b$ are in the same subset if and only if $\mathbf{y}_a - \mathbf{y}_b$ is in G_0. Let the first such element be the vector

$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \text{which yields} \quad \mathrm{G}_1 = \left\{ \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \right\}.$$

Similarly, we obtain the remaining equivalence classes,

$$G_2 = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \right\}, \quad G_3 = \left\{ \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}.$$

Note that, since $\mathbf{y^*} = \mathbf{Ty}$, the output vector of Eq. (15.24) actually specifies the state of L_2^* that corresponds to the equivalence class given by G_i.

Example Consider the linear machine L_3 given by $\{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$ over $GF(2)$ and shown in Fig. 15.16.

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix},$$

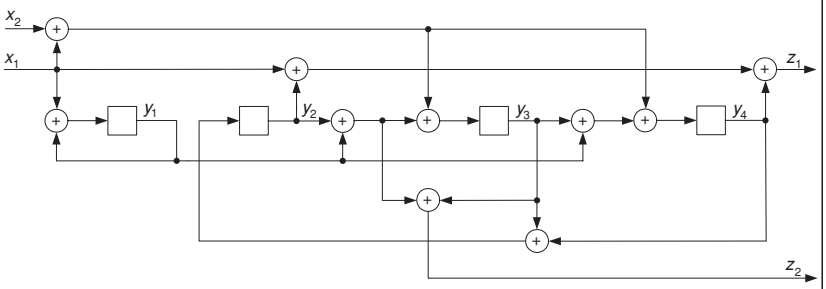


Fig. 15.16 Realization of the machine L_3.

$$\mathbf{K}_3 = \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \\ \mathbf{CA}^2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix},$$

$$\mathbf{T} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{Q} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix},$$

$$\mathbf{Q}^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{R} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}.$$

The matrix $\mathbf{Q}$ occupies the first three columns of $\mathbf{T}$ and Q^{-1} the first three rows of $\mathbf{R}$, since the linearly independent columns in $\mathbf{T}$ have been selected from positions 1, 2, and 3. We have

$$\mathbf{A}^* = \mathbf{TAR} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix},$$

$$\mathbf{B}^* = \mathbf{TB} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix},$$

$$\mathbf{C}^* = \mathbf{CR} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix},$$

$$\mathbf{D}^* = \mathbf{D} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}.$$

The reduced circuit corresponding to $\{\mathbf{A}^*, \mathbf{B}^*, \mathbf{C}^*, \mathbf{D}^*\}$ is shown in Fig. 15.17.

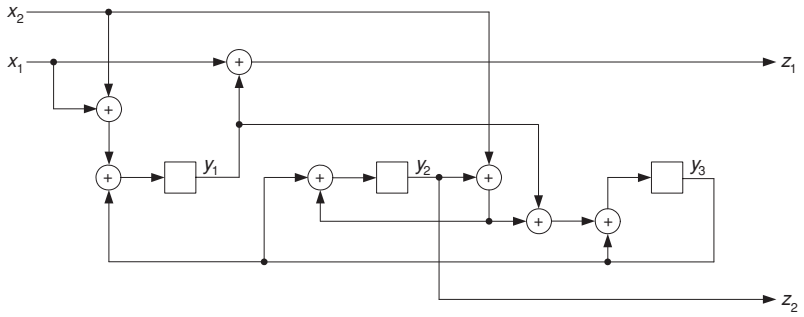


Fig. 15.17 The reduced machine L_3^*.

It is useful to note that the first three linearly independent rows of the diagnostic matrix $\mathbf{K}_3^*$ of the reduced machine L_3^* are the rows of I_3 in natural order, that is,

$$\mathbf{K}_3^* = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{matrix} \checkmark \\ \checkmark \\ \checkmark \\ \\ \\ \\ \end{matrix}$$

From Eq. (15.23) we can show that the matrix $(\mathbf{A}^*)^t$ of the reduced machine is related to the original matrix $\mathbf{A}^t$ by

$$(\mathbf{A}^*)^t = \mathbf{TA}^t\mathbf{R}$$

and that the diagnostic matrix $\mathbf{K}^*$ is related to $\mathbf{K}$ by

$$\mathbf{K}^* = \mathbf{KR}.$$

The formal proof of the above relationships is left to the reader as an exercise (see Problem 15.23). Their immediate consequence is summarized as follows.

- The first r linearly independent rows of the matrix $\mathbf{K}_r^*$ of a reduced linear machine are the rows of the identity matrix $\mathbf{I}_r$.

Applying the above results to Eq. (15.22) suggests that for an initial state $\mathbf{y}_a^* = [\mathbf{y}_1^*, \mathbf{y}_2^*, \ldots, \mathbf{y}_r^*]^T$ (where $[\mathbf{y}]^T$ denotes the transpose of $\mathbf{y}$) and under an all-0's input sequence the output values corresponding to the unit vector rows of $\mathbf{K}_{\mathrm{r}}^*$ are identical to the values $y_1^*, y_2^*, \ldots, y_r^*$. This result is of paramount importance in the identification problem of linear machines, which is discussed in the following section.

15.6 Identification of linear machines

We shall now establish certain conditions under which a reduced sequential machine will be linearly realizable. We shall determine an appropriate state assignment and define the characterizing matrices of a linear machine of the smallest dimension. We will assume that the input and output symbols of the machine are taken from $GF(p)$ and that the zero element of the field is specified. If a machine is not linearly realizable, one of several tests in the procedure will fail.

The identification procedure

From the discussion in Section 15.5 we know that a linearly realizable machine must have exactly p^k states for some integer k. Moreover, *a machine is equivalent to a linear machine if and only if its reduced form is linear.*

Let a sequential machine M have p^k states, denoted $S_a, S_b, \ldots, S_{p^k}$, and let the l-dimensional vector $\mathbf{x}$ and the m-dimensional vector $\mathbf{z}$ denote its input and output vectors, respectively. We construct for M a *distinguishing table* that contains the *output symbols generated by M in response to a sequence of* 0*'s*. The table contains p^k columns corresponding to the states of M. It is formed block by block; the ith block corresponds to the output vector $\mathbf{z}(t)$ at $t = i$. The table thus contains at most k blocks of m rows each, corresponding to the output vectors $\mathbf{z}(0), \mathbf{z}(1), \ldots, \mathbf{z}(k-1)$. The process of adding blocks to the table is terminated when, for some t, the set of rows contained in block $\mathbf{z}(t)$ is linearly dependent on the rows in preceding blocks.

As an example, we will construct the distinguishing table for the machine M_4 of Table 15.1. It is given in Table 15.2. The entries in the column headed A are 11, 01 and correspond to the output symbols of M_4 when it is initially in state A and given the input sequence 00. The construction of Table 15.2 terminates after the second block since the rows of $\mathbf{z}(1)$ are linear combinations of those of $\mathbf{z}(0)$. We shall subsequently denote the distinguishing table by U.

Table 15.1 Machine M_4

	NS, z_1z_2	
PS	$x=0$	$x=1$
A	B, 11	D, 01
B	A, 01	C, 11
C	C, 10	A, 00
D	D, 00	B, 10

Table 15.2 Distinguishing table for M_4

	A	B	C	D
$\mathbf{z}(0)$	1	0	1	0
	1	1	0	0
$\mathbf{z}(1)$	0	1	1	0
	1	1	0	0

Since the input and output symbols of M_4 are limited to 0 and 1, the linear realization has to be over $GF(2)$. The first test is based on the fact that, for every linear machine, the all-0's sequence is a distinguishing sequence. If M is reduced then the columns of U must be distinct, since otherwise there would be two or more states in M that are indistinguishable under the all-0's sequence, and M would not be linear. Clearly, Table 15.2 passes this test.

Let U^* be the table consisting of the first r linearly independent rows of U, and let S_i denote the ith column of U^*. Assuming that a linear realization of M is possible, let the states $A, B, \ldots$ of M correspond to the state vectors $\mathbf{y}_a, \mathbf{y}_b, \ldots$ of its linear realization L. This is accomplished by selecting the p^k columns of U^* as the state assignment for the p^k states of L. For the machine L_4, which is to be the linear realization of M_4, we have

$$\mathbf{y}_a = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \mathbf{y}_b = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \mathbf{y}_c = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{y}_d = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

In the above step, it has been implicitly assumed that if a linear realization exists, its state assignment is given by U^*. This assertion follows directly from the result of the preceding section, in which it was shown that, under an all-0's input sequence, the output values corresponding to the r linearly independent rows of $\mathbf{K}_r^*$ are identical to the state assignment given by $(y_1^*, y_2^*, \ldots, y_r^*)$. In addition, since the rows of U^* are the linearly independent output vectors associated with the states of L, they are also equal to the state assignment of L.

In order to obtain the set of characterizing matrices $\{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$ of L, we select r linearly independent columns from U^*, corresponding to the r state vectors of L, and form an $r \times r$ matrix $\mathbf{v}$ such that

$$\mathbf{v} = [\mathbf{y}_a \quad \mathbf{y}_b \quad \cdots \quad \mathbf{y}_r].$$

From Eq. (15.13), we find that the next-state function of L under input symbols 0 is

$$[\mathbf{Y}_a^0 \quad \mathbf{Y}_b^0 \quad \cdots \quad \mathbf{Y}_r^0] = \mathbf{Av},$$

where $\mathbf{Y}_i^0$ denotes the 0-successor of $\mathbf{y}_i$. Since $\mathbf{v}$ is nonsingular, we can write

$$\mathbf{A} = [\mathbf{Y}_a^0 \quad \mathbf{Y}_b^0 \quad \cdots \quad \mathbf{Y}_r^0]\mathbf{v}^{-1}. \tag{15.25}$$

If all r unit vectors appear in U^* then $\mathbf{v}$ can be chosen as $\mathbf{I}_r$, which yields $\mathbf{v} = \mathbf{v}^{-1}$, and so Eq. (15.25) is reduced to

$$\mathbf{A} = [\mathbf{Y}_a^0 \quad \mathbf{Y}_b^0 \quad \cdots \quad \mathbf{Y}_r^0]. \tag{15.26}$$

Whenever the number of states $p^k = p^r$, i.e., $k = r$, $\mathbf{v}$ can be specified as $\mathbf{I}_r$.

Similarly, from Eq. (15.14) and for $\mathbf{x}(t) = \mathbf{0}$, we find that

$$[\mathbf{z}_a^0 \quad \mathbf{z}_b^0 \quad \cdots \quad \mathbf{z}_r^0] = \mathbf{C}\mathbf{v},$$

where $\mathbf{z}_i^0$ denotes the output symbol produced by L when in the state $\mathbf{y}_i$ and excited by the input symbol $\mathbf{x} = \mathbf{0}$. Thus

$$\mathbf{C} = [\mathbf{z}_a^0 \quad \mathbf{z}_b^0 \quad \cdots \quad \mathbf{z}_r^0]\mathbf{v}^{-1} \tag{15.27}$$

and so, when $\mathbf{v} = \mathbf{I}_r$,

$$\mathbf{C} = [\mathbf{z}_a^0 \quad \mathbf{z}_b^0 \quad \cdots \quad \mathbf{z}_r^0]. \tag{15.28}$$

In order to obtain $\mathbf{B}$ and $\mathbf{D}$, let us denote a unit input vector as $\mathbf{u}_i$, where the ith component of $\mathbf{u}_i$ is 1 and all other components are 0's. From Eq. (15.13) we obtain

$$\mathbf{B}\mathbf{x} = \mathbf{Y} - \mathbf{A}\mathbf{y}.$$

In order to obtain $\mathbf{B}$, we select some state $\mathbf{y}_i$ (preferably the zero state if it exists in U^*) and specify $\mathbf{B}$ in terms of the constraints imposed on it by $\mathbf{y}_i$ and the unit input vectors. Clearly, such a process does not guarantee that the selection of another $\mathbf{y}_j$ will specify the same $\mathbf{B}$ matrix, unless the machine being identified is indeed linear. For the time being, we shall specify a set of characterizing matrices and will check them for all possible input and state combinations at the end of the test.

Let the input consist of the unit vectors

$$\mathbf{u} = [\mathbf{u}_1 \quad \mathbf{u}_2 \quad \cdots \quad \mathbf{u}_l].$$

The next-state vector $\mathbf{Y}_i^{u_j}$ denotes the u_j-successor of y_i. Thus,

$$\mathbf{Y}_i^u = [\mathbf{Y}_i^{u_1} \quad \mathbf{Y}_i^{u_2} \quad \cdots \quad \mathbf{Y}_i^{u_l}]$$

and

$$\mathbf{B}\mathbf{u} = \mathbf{Y}_i^u - \mathbf{A}\mathbf{y}_i$$

or

$$\mathbf{B} = [\mathbf{Y}_i^u - \mathbf{A}\mathbf{y}_i] \quad \mathbf{u}^{-1}. \tag{15.29}$$

Since $\mathbf{u}$ generally consists of unit vectors, when $\mathbf{y}$ is the zero state Eq. (15.29) reduces to

$$\mathbf{B} = [\mathbf{Y}_i^{u_1} \quad \mathbf{Y}_i^{u_2} \quad \cdots \quad \mathbf{Y}_i^{u_l}]. \tag{15.30}$$

Similarly, from Eq. (15.14) we obtain

$$\mathbf{D} = \{[\mathbf{z}_i^{u_1} \quad \mathbf{z}_i^{u_2} \quad \cdots \quad \mathbf{z}_i^{u_l}] - \mathbf{A}\mathbf{y}_i\}\mathbf{u}^{-1}, \tag{15.31}$$

where $\mathbf{z}_i^{u_j}$ is the output vector associated with the transition from $\mathbf{y}_i$ under an input $\mathbf{u}_j$. In analogy with Eq. (15.30) the reduced equation is

$$\mathbf{D} = [\mathbf{z}_i^{u_1} \quad \mathbf{z}_i^{u_2} \quad \cdots \quad \mathbf{z}_i^{u_l}]. \tag{15.32}$$

Returning to machine M_4, we make the specification

$$\mathbf{v} = \begin{bmatrix} \mathbf{y}_c & \mathbf{y}_b \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \mathbf{I}_2.$$

From Eqs. (15.26) and (15.28), we obtain

$$\mathbf{A} = \begin{bmatrix} \mathbf{Y}_c^0 & \mathbf{Y}_b^0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} \mathbf{z}_c^0 & \mathbf{z}_b^0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

The only unit input vector is $\mathbf{u} = [1]$, and hence $\mathbf{Y}_i^1$ is the 1-successor of $\mathbf{y}_i$. Since the zero state is contained in U^*, let $\mathbf{y}_i = \mathbf{y}_d$; then, by Eqs. (15.30) and (15.32), we obtain

$$\mathbf{B} = \begin{bmatrix} \mathbf{Y}_d^1 \end{bmatrix} = [\,\mathbf{y}_b] = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} \mathbf{z}_d^1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

The state and output equations are

$$\mathbf{Y}(t) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \mathbf{y}(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \mathbf{x}(t),$$

$$\mathbf{z}(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \mathbf{y}(t) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \mathbf{x}(t).$$

The final test is to verify that the above equations indeed represent the machine M_4 under *all* input and state combinations. This is accomplished by verifying each state transition and its corresponding output symbol. For example, substituting $\mathbf{y}_a$ for A and $\mathbf{0}$ for $\mathbf{x}(t)$, the machine should go to the state $\mathbf{y}_b$ and produce the output symbol 11, corresponding to the entry B, 11 in column 0, row A, in Table 15.1. Indeed,

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} [0] = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \rightarrow \quad \mathbf{y}_b,$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} [0] = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \rightarrow \quad \mathbf{z}_a^0.$$

The characterizing matrices are thus verified, and the linear realization of Fig. 15.18 results.

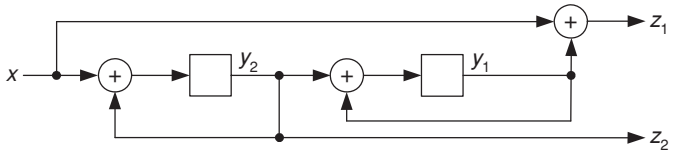


Fig. 15.18 The machine L_4.

Example The machine M_5 and its distinguishing table are given in Tables 15.3 and 15.4, respectively. The "checked" rows are linearly independent, and since U^* contains all eight possible 3-tuples, the identification procedure is continued.

Table 15.3 Machine M_5

	NS, z_1z_2	
PS	$x=0$	$x=1$
A	A, 00	E, 10
B	A, 10	E, 00
C	B, 11	F, 01
D	B, 01	F, 11
E	C, 01	G, 11
F	C, 11	G, 01
G	D, 10	H, 00
H	D, 00	H, 10

Table 15.4 Distinguishing table for M_5

	A	B	C	D	E	F	G	H	
$\mathbf{z}(0)$	0	1	1	0	0	1	1	0	✓
	0	0	1	1	1	1	0	0	✓
$\mathbf{z}(1)$	0	0	1	1	1	1	0	0	
	0	0	0	0	1	1	1	1	✓
$\mathbf{z}(2)$	0	0	0	0	1	1	1	1	
	0	0	0	0	0	0	0	0	

Therefore, select

$$\mathbf{v} = \begin{bmatrix} \mathbf{y}_b & \mathbf{y}_d & \mathbf{y}_h \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \mathbf{I}_3.$$

From Eqs. (15.26) and (15.28), we obtain

$$\mathbf{A} = \begin{bmatrix} \mathbf{Y}_b^0 & \mathbf{Y}_d^0 & \mathbf{Y}_h^0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{z}_b^0 & \mathbf{z}_d^0 & \mathbf{z}_h^0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Setting $\mathbf{u} = [1]$ and $\mathbf{y}_i = \mathbf{y}_a = \mathbf{0}$, Eqs. (15.30) and (15.32) yield

$$\mathbf{B} = \left[\mathbf{Y}_a^1\right] = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{D} = \left[\mathbf{z}_a^1\right] = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Thus

$$\mathbf{Y}(t) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \mathbf{y}(t) + \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \mathbf{x}(t),$$

$$\mathbf{z}(t) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \mathbf{y}(t) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \mathbf{x}(t).$$

The matrices are verified as corresponding to M_5, and their linear realization is given in Fig. 15.19.

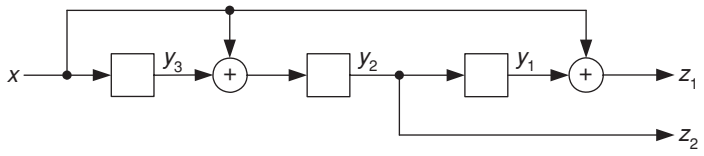


Fig. 15.19 The machine L_5.

Example As another example, consider the four-stage up–down Gray-code counter of Table 15.5, whose distinguishing table is given in Table 15.6.

Table 15.5 The machine M_6

	NS		
PS	$x = 0$	$x = 1$	$z_1 z_2$
A	B	D	00
B	C	A	01
C	D	B	11
D	A	C	10

Table 15.6 Distinguishing table for M_6

	A	B	C	D	
$\mathbf{z}(0)$	0	0	1	1	✓
	0	1	1	0	✓
$\mathbf{z}(1)$	0	1	1	0	
	1	1	0	0	✓
$\mathbf{z}(2)$	1	1	0	0	
	1	0	0	1	

The state assignment is given by

$$\mathbf{y}_a = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{y}_b = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{y}_c = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{y}_d = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

Note that although M_6 has only four states, its minimal linear realization has a third dimension; that is, if M_6 is linearly realizable then it is realizable as a submachine of an eight-state linear machine. Note also that $\mathbf{v}$ cannot be chosen as the identity matrix, and the zero state $\mathbf{y}_i = \mathbf{0}$ is not contained in the state assignment. Consequently, the simplified equations cannot be used, and matrix inversion cannot be avoided. Let

$$\mathbf{v} = \begin{bmatrix} \mathbf{y}_d & \mathbf{y}_b & \mathbf{y}_a \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}; \quad \text{then} \quad \mathbf{v}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

From Eqs. (15.25) and (15.27), we obtain

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \mathbf{v}^{-1} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix},$$

$$\mathbf{C} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \mathbf{v}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Let $\mathbf{y}_i^1 = \mathbf{y}_a$. Then from Eq. (15.29) we obtain

$$\mathbf{B} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} - \mathbf{A} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

The minimum-dimensional linear circuit realizing the counter is shown in Fig. 15.20.

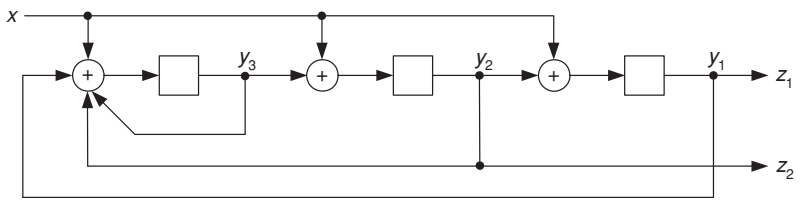


Fig. 15.20 Linear realization of the Gray-code counter.

15.7 Application of linear machines to error correction

The availability of analysis and synthesis techniques for linear machines and their economical realization by means of shift registers have made them widely

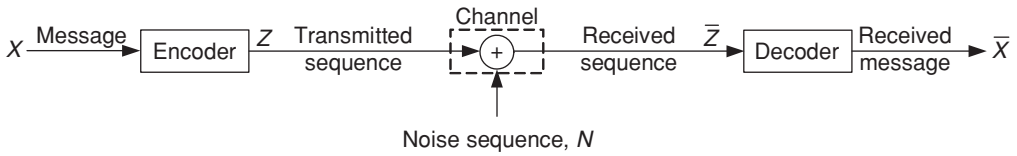


Fig. 15.21 A model for a communication system.

applicable in communication and digital computation. Linear machines are particularly useful in computations involving the multiplication and division of polynomials and in error detection and correction. In this section we describe in detail how they can be used in a simple error-correcting coding scheme. For a more complete survey of coding and digital computation applications, the reader is referred to Peterson [15] and Gill [9].

Consider the communication-system model shown in Fig. 15.21. The *message*, denoted X, consists of a sequence over $GF(p)$ of length n. The *encoder*, whose transfer function is T, transforms the message into another sequence over $GF(p)$ of length n. This sequence is referred to as the *transmitted sequence* and is designated Z, where $Z = TX$. The sequence Z is transmitted through a noisy *channel*, whose output sequence $\bar{Z}$ is called the *received sequence*. In the channel, a *noise sequence* over $GF(p)$, denoted N, is added to the transmitted sequence, so that the received sequence is equal to

$$\begin{aligned}\bar{Z} &= Z + N \\ &= TX + N.\end{aligned}$$

The *decoder*, whose transfer function is T^{-1}, processes the received sequence and produces a sequence $\bar{X}$ such that

$$\begin{aligned}\bar{X} &= T^{-1}\bar{Z} \\ &= T^{-1}(TX + N) \\ &= X + T^{-1}N.\end{aligned}$$

If the noise sequence is equal to zero, that is, $N = 0$, then the *received message* $\bar{X}$ is a replica of the original message X, that is, $X = \bar{X}$. If the noise sequence is different from zero then the received message $\bar{X}$ consists of the modulo-p sum of the original message X and the response $T^{-1}N$ of the decoder to the noise sequence.

As an illustration of the error-correction procedure, let us analyze in detail the communication system shown in Fig. 15.22, where the encoder's transfer function is given by $T = 1 + D^2 + D^3$ and the message as well as the noise are over $GF(2)$. We assume that the noise sequence contains only a single nonzero digit; i.e., the communication system is *single-error-correcting*. Suppose that a seven-bit message X is to be transmitted, where the first four digits are the *information digits* and the remaining three digits are the *checking digits*. The checking digits in X are always 0's. Consequently, if $\bar{X}$ is received with three 0's in the last three positions then it means that no noise is present in the channel and $\bar{X}$ is an identical replica of X. If, however, the received message $\bar{X}$ contains nonzero digits in the last three positions, this indicates that an error

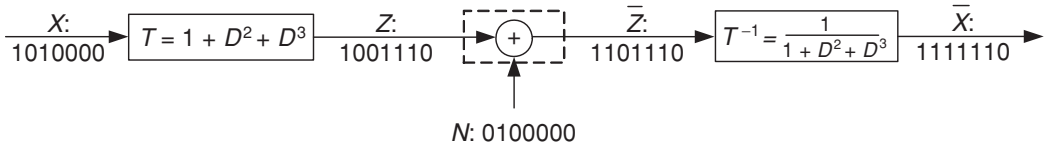


Fig. 15.22 An example of a linear single-error-correcting scheme.

has occurred during transmission and an error-correcting procedure must be employed to recover the original message.

When an error occurs, it is necessary to obtain the sequence $T^{-1}N$ and subtract it from the received message $\bar{X}$. To obtain $T^{-1}N$, we observe that since the last three digits of X were originally 0's then the last three digits of $\bar{X}$ must consist only of digits of $T^{-1}N$, without any contribution from X. In fact, if only a single error occurred at time t then the sequence $T^{-1}N$ is simply the response of decoder T^{-1} to a unit impulse occurring at t. Therefore, the checking digits of $\bar{X}$ consist of a subsequence of three digits of the impulse response of T^{-1}. (Clearly, if the error occurs in one of the checking digits, say in the second checking digit, then the first digit will be a zero and the remaining two checking digits will be the first two digits of the impulse response of T^{-1}.)

The decoder is chosen so that its impulse response has a maximal period of seven digits. This ensures that, by observing the subsequence contained in the last three digits of $\bar{X}$, we can determine uniquely the entire sequence $T^{-1}N$. Since a maximal impulse response contains all seven possible combinations of three successive nonzero digits, each noise impulse corresponds to only one pattern of checking digits and thus its location can be uniquely determined.

As an example, suppose that the sequence 1010000 is to be transmitted by means of the communication system of Fig. 15.22. The transmitted sequence Z is found to be 1001110. If an error occurs in the second digit, the received sequence $\bar{Z}$ will be 1101110. Since the impulse response of the decoder, whose transfer function is $T^{-1} = (1 + D^2 + D^3)^{-1}$, is 1011100, the received message $\bar{X}$ is equal to 1111110. The checking digits of $\bar{X}$ are identical to the fourth, fifth, and sixth digits of the impulse response. Consequently, we may conclude that the noise impulse has occurred in the second information digit. The sequence $T^{-1}N$ is thus found to be 0101110, and it may now be added (the same as subtracting modulo 2) to $\bar{X}$ to obtain the original message X, i.e.,

Decoder's impulse response: 1 0 1 1 1 0 0

$\bar{X}$:	1	1	1	1	1	1	0	+
$T^{-1}N$:	0	1	0	1	1	1	0	
X:	1	0	1	0	0	0	0	

In a similar manner, the reader can verify that if the message 1110000 is transmitted by means of the system of Fig. 15.22, and the noise N is given by 0010000, then the received message would be 1100111. The checking digits

contain the third, fourth, and fifth digits of the decoder's impulse response. Consequently, $T^{-1}N$ is equal to 0010111, and the message X can be reconstructed.

To obtain single-error correction for messages over $GF(2)$ containing m information digits and k checking digits, we need a decoder whose impulse response is of length $m + k$, with each string of k successive digits different from every other subsequence of length k. Such an impulse response can be obtained from a decoder whose transfer function is of degree k and whose impulse response is maximal, i.e., of length $m + k = 2^k - 1$. If the last k digits of received message $\bar{X}$ are not zeros then the sequence $T^{-1}N$ must be subtracted from $\bar{X}$. This can be accomplished by shifting $\bar{X}$ over the decoder's impulse response until the last k digits of $\bar{X}$ match a corresponding subsequence of the impulse response. This is always possible since the impulse response contains every nonzero subsequence of length k. The modulo-2 sum of $\bar{X}$ and the digits of the impulse response appearing directly below it yield the original message X.

Appendix 15.1 Basic properties of finite fields[5]

A set R is said to form a *ring* if two operations, addition and multiplication, are defined for every pair of elements in R, and if it satisfies the following postulates.

1. *Closure* For every a and b in R, $a + b$ and ab are in R.
2. *Associativity* For every a, b, and c in R, $(a + b) + c = a + (b + c)$ and $(ab)c = a(bc)$.
3. The set R contains a *unique zero element*, denoted 0, such that, for every a in R, $a + 0 = 0 + a = a$.
4. To each a in R, there corresponds a unique element $-a$ in R such that $a + (-a) = (-a) + a = 0$; $-a$ is called the *inverse* of a.
5. *Distributivity* Multiplication distributes over addition; that is, $a(b + c) = ab + ac$, for all a, b, and c in R.
6. *Commutativity* For all a and b in R, $a + b = b + a$.

If multiplication is also commutative, i.e., $ab = ba$, R is said to be a *commutative* ring.

Example The set of integers $\{0, 1, \ldots, p - 1\}$ under modulo-p addition and multiplication operations forms a commutative ring. (Note that modulo p means that a is equal to b whenever $a - b$ is a multiple of p). The definition of modulo-4 operations is shown in Table A15.1.

[5] This is only a short summary of several definitions and results in the area of fields. For a more complete coverage, the reader is referred to any book on algebra.

Table A15.1 Addition and multiplication modulo 4

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

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

The set F is said to be a *field* if it is a commutative ring and, in addition, satisfies the following two postulates.

1. There is a unique nonzero element 1 in F such that $a1 = a$ for every a in F.
2. To each nonzero a in F, there corresponds a unique element a^{-1} (or $1/a$) in F such that $aa^{-1} = 1$.

The set of real numbers and the set of complex numbers each forms an infinite field. Fields containing a finite number of elements are usually called *finite fields*.

Example The modulo-4 ring defined in Table A15.1 is not a field, since the element 2 does not have a multiplicative inverse; that is, the equation $2a = 1$ does not have a solution for a, as can be seen from the defining table. However, the equation $2a = 2$ (modulo 4) has two solutions, $a = 1$ and $a = 3$.

The above example illustrates the reason for restricting our discussion of linear machines to modulo p of prime numbers: multiplication by numbers that are not prime to the modulo may be irreversible and, consequently, may not preserve information. It can be shown that if p is a prime integer, then the ring of integers, modulo p, forms a field. This finite field is called a *Galois field* and is denoted $GF(p)$.

Example The set of integers $\{0, 1, 2\}$ and the operations defined in Table A15.2 form the finite field $GF(3)$.

Table A15.2 Modulo-3 operations

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

·	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

Any Galois field with prime characteristic p contains exactly p^k elements, for some integer k. This field is denoted $GF(p^k)$. It can also be shown that, for

any finite field, there exists a prime integer p and a positive integer k such that the given field is equivalent to $GF(p^k)$.

In this chapter the fields were defined over $GF(p)$, where p is a prime. The theory and results obtained can be generalized to include linear machines defined over any finite field. It can be shown [17] that there exists an equivalence between a linear machine defined over any finite field and a linear machine defined over $GF(p)$. Consequently, any linear machine defined over any finite field can be synthesized by the techniques developed for machines defined over $GF(p)$, where p is a prime integer.

Appendix 15.2 The Euclidean algorithm

The Euclidean algorithm provides a procedure for obtaining the greatest common divisor of two polynomials over a field F.

Let $P(D)/Q(D)$ be a rational polynomial of the following form:

$$\frac{P(D)}{Q(D)} = \frac{a_0 + a_1 D + \cdots + a_m D^m}{b_0 + b_1 D + \cdots + b_n D^n},$$

where the degree of $P(D)$ is smaller than that of $Q(D)$. (The degree of a polynomial $P(D)$ is the greatest i such that $a_i \neq 0$.) The Euclidean algorithm is based on the result that every rational polynomial can be divided in a unique manner such that

$$Q(D) = q(D)P(D) + r(D).$$

When the remainder $r(D) = 0$, $P(D)$ is said to divide $Q(D)$. To find the greatest common divisor, we use successive division as follows:

$$\begin{aligned} Q(D) &= q_1(D)P(D) + r_1(D), \\ P(D) &= q_2(D)r_1(D) + r_2(D), \\ r_1(D) &= q_3(D)r_2(D) + r_3(D), \\ &\vdots \\ r_{i-2}(D) &= q_i(D)r_{i-1}(D). \end{aligned}$$

Then $r_{i-1}(D)$ is the greatest common divisor of $P(D)$ and $Q(D)$.

Example Determine the greatest common divisor for the polynomial

$$T(D) = \frac{P(D)}{Q(D)} = \frac{1 + D + D^4 + D^6}{D + D^3 + D^4 + D^6 + D^8 + D^9} \quad \text{(over } GF(2)\text{)}.$$

Proceeding by successive division,

$$
\begin{array}{r|l}
D^6 + D^4 + D + 1 & D^3 + D^2 + D \\
\hline
 & D^9 + D^8 + D^6 + D^4 + D^3 + D \\
 & \underline{D^9 + D^7 + D^4 + D^3} \\
 & D^8 + D^7 + D^6 + D \\
 & \underline{D^8 + D^6 + D^3 + D^2} \\
 & D^7 + D^3 + D^2 + D \\
 & \underline{D^7 + D^5 + D^2 + D} \\
 & D^5 + D^3 \quad \longleftarrow \text{ determination of } r_1(D)
\end{array}
$$

$$
\begin{array}{r|l}
D^5 + D^3 & D \\
\hline
 & D^6 + D^4 + D + 1 \\
 & \underline{D^6 + D^4} \\
 & D + 1 \quad \longleftarrow \text{ determination of } r_2(D)
\end{array}
$$

$$
\begin{array}{r|l}
D + 1 & D^4 + D^3 \\
\hline
 & D^5 + D^3 \\
 & \underline{D^5 + D^4} \\
 & D^4 + D^3 \\
 & \underline{\underline{D^4 + D^3}} \quad \longleftarrow \; r_3(D) = 0
\end{array}
$$

Since $r_3(D) = 0$, $r_2(D) = D + 1$ is the greatest common divisor. To find the reduced polynomial, it is necessary to divide $P(D)$ and $Q(D)$ by $D + 1$. This division yields

$$T(D) = \frac{1 + D^4 + D^5}{D + D^2 + D^4 + D^5 + D^8}.$$

Notes and references

Linear machines were first investigated by Huffman in 1956 [13]. This original work, which was restricted to inert machines, was later expanded by several people, notably Cohn [3, 4], Elspas [7], Friedland [8], Hartmanis [10], and Stern and Friedland [17]. The problem of identifying linear machines was treated by numerous authors, among them Brzozowski and Davis [2], Davis and Brzozowski [6] and Hartmanis [11]. The most general minimization and identification procedure is due to Cohn and Even [5], whose approach has been followed in this chapter. Other aspects of linear machines were studied by Booth [1], Pugsley [16], and Zierler [18]. The application of linear machines to error-correcting codes is due to Huffman [12] and Peterson [15]. A good collection of papers on linear machines is available in Kautz [14]. One of the best general treatments of linear machines can be found in the book by Gill [9].

[1] Booth, T. L.: "An analytic representation of signals in sequential networks," in *Proc. Symp. Mathematical Theory of Automata*, vol. 12, pp. 301–340, Polytechnic Institute of Brooklyn, New York, 1963.
[2] Brzozowski, J. A., and W. A. Davis: "On the linearity of autonomous sequential machines," *Trans. IEEE*, vol. EC-13, pp. 673–679, 1964.
[3] Cohn, M.: "Controllability in linear sequential networks," *Trans. IRE*, vol. CT-9, pp. 74–78, 1962.
[4] Cohn, M.: "Properties of linear machines," *J. Assoc. Computing Machinery*, vol. 11, pp. 296–301, 1964.
[5] Cohn, M., and S. Even: "Identification and minimization of linear machines," *Trans. IEEE*, vol. EC-14, pp. 367–376, 1965.
[6] Davis, W. A., and J. A. Brzozowski: "On the linearity of sequential machines," *Trans. IEEE*, vol. EC-15, pp. 21–29, 1966.
[7] Elspas, B.: "The theory of autonomous linear sequential networks," *Trans. IRE*, vol. CT-6, pp. 45–60, 1959.
[8] Friedland, B.: "Linear modular sequential circuits," *Trans. IRE*, vol. CT-6, pp. 61–68, 1959.
[9] Gill, A.: *Linear Sequential Circuits*, McGraw-Hill, New York, 1967.
[10] Hartmanis, J.: "Linear multivalued sequential coding networks," *Trans. IRE*, vol. CT-6, pp. 69–74, 1959.
[11] Hartmanis, J.: "Two tests for the linearity of sequential machines," *Trans. IEEE*, vol. EC-14, pp. 781–786, 1965.
[12] Huffman, D. A.: "A linear circuit viewpoint of error-correcting codes," *Trans. IRE*, vol. IT-2, pp. 20–28, 1956.
[13] Huffman, D. A.: "The synthesis of linear sequential coding networks," in C. Cherry (ed.), *Information Theory*, pp. 77–95, Academic Press, New York, 1956.
[14] Kautz, W. H. (ed.): *Linear Sequential Switching Circuits: Selected Technical Papers*, Holden-Day, 1965.
[15] Peterson, W. W.: *Error-correcting Codes*, M.I.T. Press, Cambridge MA, 1961.
[16] Pugsley, J. H.: "Sequential functions and linear sequential machines," *Trans. IEEE*, vol. EC-14, pp. 376–382, 1965.
[17] Stern, T. E., and B. Friedland: "The linear modular sequential circuit generalized," *Trans. IRE*, vol. CT-8, pp. 79–80, 1961.
[18] Zierler, N.: "Linear recurring sequences," *J. Soc. Ind. Appl. Math.*, vol. 7, pp. 31–48, 1959.

Problems

Problem 15.1. A *combinational linear circuit* is a circuit constructed only of modulo-p adders and multipliers. The block diagram in Fig. P15.1 represents a combinational linear circuit over $GF(2)$. The circuit outputs can be expressed as

$$z_a = x_a,$$
$$z_b = x_a + x_b,$$
$$z_c = x_b + x_c.$$

(a) Show the circuit diagram.

(b) Find the output sequences in response to the following input sequences:

x_a :	0	1	0	1	1	1	1	0	0	0	1	0	1	1
x_b :	1	1	0	1	0	0	0	0	1	0	1	1	0	1
x_c :	0	0	1	1	0	1	1	0	1	0	0	0	0	1

(c) Design the inverse of this circuit; i.e., express the inputs as functions of the outputs and show the inverse circuit.

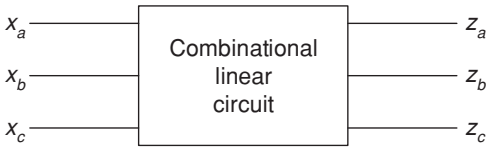


Fig. P15.1

Problem 15.2

(a) Determine the transfer function of the shift register shown in Fig. P15.2.
(b) Find its null sequence and show that it is maximal.
(c) Find the inverse machine.

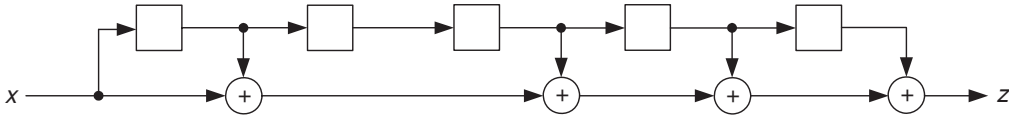


Fig. P15.2

Problem 15.3. For each of the following polynomials over $GF(2)$,

$$z_1 = x + D^3x + D^4x, \quad z_2 = x + D^2x + D^4x + D^5x:$$

(a) show the corresponding linear circuit and its inverse;
(b) find the null sequence and determine whether it is maximal;
(c) utilize the impulse response to determine the response of each circuit to the input sequence 000001101.

Problem 15.4. Show the state diagram of the linear machine whose transfer function is $T = 1 + D + D^3$.

Problem 15.5. Prove that the two circuits over $GF(3)$ of Fig. P15.5 are equivalent.

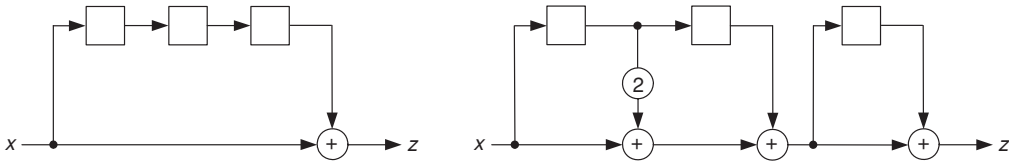


Fig. P15.5

Problem 15.6. Prove that the two circuits over $GF(16)$ of Fig. P15.6 have the same transfer functions. (Note that the use of feedback allows us in this case to construct a machine whose output symbol depends on input symbols three time units in the past, by using just a single delay element.)

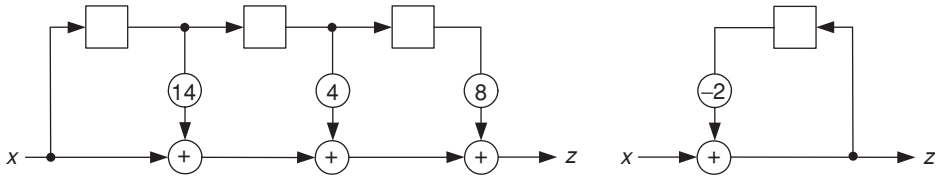


Fig. P15.6

Problem 15.7. Determine the null sequence of the linear machine over $GF(3)$ whose transfer function is $T = 2 + D^2 + 2D^3$. Prove that it is a maximal sequence.

Problem 15.8. Prove that the delay polynomial $T(D) = a_0 + a_1 D + \cdots + a_k D^k$ has a linear inverse that decodes without a delay if and only if $T(D)$ has a nonzero constant term that is relatively prime to p.

Hint: Assume initially $a_0 = 1$. Expand $1/T(D)$ into the form

$$\frac{1}{T(D)} = \frac{1}{1 + \sum_1^n a_i D^i} = 1 - \sum_1^n a_i D^i + \left(\sum_1^n a_i D^i\right)^2 - \cdots$$

Problem 15.9. Figure P15.9 shows an inert linear machine over $GF(3)$. Prove that its transfer function is

$$T = \frac{z}{x} = \frac{2D + 2D^2 + D^3}{1 + D^2}.$$

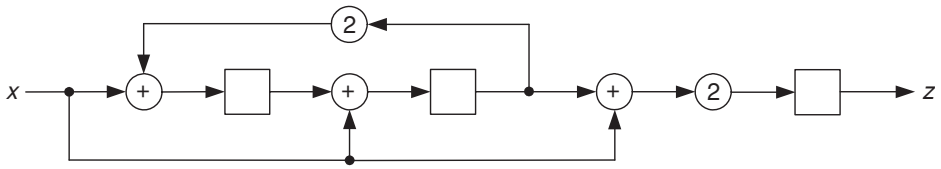


Fig. P15.9

Problem 15.10

(a) Prove that the transfer function of the inert linear machine of Fig. P15.10 is given by

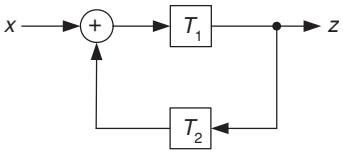
$$T = \frac{z}{x} = \frac{T_1}{1 - T_1 T_2},$$

where T_1 and T_2 are transfer functions of the individual submachines.

(b) Use the result of part (a) to find the transfer function of the machine in Fig. P15.9.

Hint: In part (b), determine first the direct paths through which the input signal can reach the output terminal.

Fig. P15.10



Problem 15.11

(a) Determine the transfer function of the linear machine over $GF(2)$ shown in Fig. P15.11 and find its impulse response. Assume that it is initially inert.

(b) Prove that its state table is isomorphic to Table P15.11.

Fig. P15.11

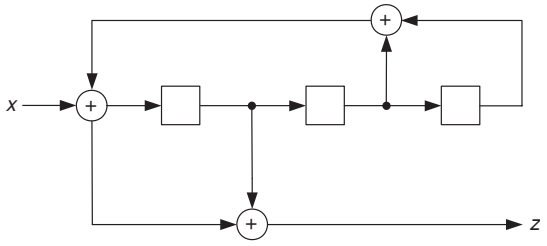


Table P15.11

	NS, z	
PS	$x = 0$	$x = 1$
A	$A, 0$	$E, 1$
B	$E, 1$	$A, 0$
C	$F, 1$	$B, 0$
D	$B, 0$	$F, 1$
E	$C, 1$	$G, 0$
F	$G, 0$	$C, 1$
G	$H, 0$	$D, 1$
H	$D, 1$	$H, 0$

Problem 15.12. For each of the following transfer functions,

$$T_1 = \frac{1 + D^2}{1 + D + D^3} \qquad \text{over } GF(2),$$

$$T_2 = \frac{D^2}{2D^2 + D + 1} \qquad \text{over } GF(3),$$

(a) show the corresponding network;

(b) find its impulse response;

(c) determine whether it is invertible and, if it is, show the inverse.

Problem 15.13. Given the following transfer function over $GF(2)$,

$$T = \frac{D^{10} + D^9 + D^8 + D^7 + D}{D^7 + D^4 + D^2 + D + 1},$$

(a) determine by means of the Euclidean algorithm the greatest common divisor of the numerator and denominator, and simplify the function;

(b) show a minimal chain realization, using no more than eight delay elements.

Problem 15.14. Show minimal realizations of the transfer function below and of its inverse.

$$T = \frac{1 + D + 2D^2 + D^3}{1 + D + D^3 + 2D^4} \qquad \text{over } GF(3).$$

Problem 15.15. Design a four-dimensional linear machine over $GF(2)$ whose impulse response is

$$h = 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad (1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0) \cdots$$

(The sequence in parentheses repeats itself thereafter.)

Problem 15.16. Show the linear circuit over $GF(2)$ whose characterizing matrices are

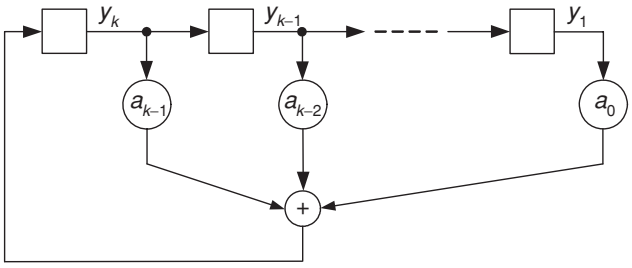
$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

Problem 15.17

(a) Find the characteristic matrix $\mathbf{A}$ that is realized by the internal circuit of Fig. P15.17.

(b) Determine the transpose of the matrix $\mathbf{A}$ in part (a), and show a circuit that realizes the transposed matrix.

Fig. P15.17



Problem 15.18

(a) Prove that a linear machine $\{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$ is μ-definite if and only if μ is the least integer such that $\mathbf{A}^\mu = 0$.

(b) Prove that if a k-dimensional linear machine is μ-definite then $\mu \leq k$.
Hint: See [4].

Problem 15.19

(a) Design the linear circuit over $GF(2)$ whose characterizing matrices are

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix},$$

$$\mathbf{C} = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}.$$

(b) Minimize the machine of part (a), and show that it is independent of x_2.

Problem 15.20

(a) Minimize the linear machine over $GF(2)$ given by the following characterizing matrices:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix}, \quad \mathbf{D} = [0].$$

(b) For each state of the reduced machine, show the equivalent states of the original machine.

Problem 15.21

(a) Design the linear circuit over $GF(2)$ whose characterizing matrices are

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

(b) Prove that no reduction in the machine dimension is possible but that the reduction procedure can be applied to obtain an equivalent machine $\{\mathbf{A}^*, \mathbf{B}^*, \mathbf{C}^*, \mathbf{D}^*\}$ that is realizable with a single modulo-2 adder.

Problem 15.22

(a) Given a linear machine $L = \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$ and a nonsingular matrix $\mathbf{G}$, prove that the state $\mathbf{y}$ of L is equivalent to the state $\bar{\mathbf{y}} = \mathbf{G}\mathbf{y}$ of $\bar{L}$, where $\bar{L}$ is the linear machine characterized by

$$\bar{\mathbf{A}} = \mathbf{G}\mathbf{A}\mathbf{G}^{-1}, \quad \bar{\mathbf{B}} = \mathbf{G}\mathbf{B}, \quad \bar{\mathbf{C}} = \mathbf{C}\mathbf{G}^{-1}, \quad \bar{\mathbf{D}} = \mathbf{D}.$$

(b) Prove that the machines L and $\bar{L}$ are isomorphic.

Problem 15.23

(a) Prove that, for all $t \geq 0$,

$$(\mathbf{A}^*)^t = \mathbf{T}\mathbf{A}^t\mathbf{R},$$

where $\mathbf{A}^*$ is the characteristic matrix of the reduced machine, defined in Eq. (15.23). *Hint:* Prove the assertion for $t = 0$ and use induction on t.

(b) Use the result of part (a) to prove that the diagnostic matrix $\mathbf{K}^*$ of the reduced machine is related to $\mathbf{K}$ by

$$\mathbf{K}^* = \mathbf{K}\mathbf{R}.$$

(c) Prove that if $\mathbf{T}^*$ is the $r \times r$ matrix consisting of the first r linearly independent rows of $\mathbf{K}_r^*$ of a reduced linear machine then $\mathbf{T}^* = \mathbf{I}_r$, where $\mathbf{I}_r$ is the identity matrix.

Problem 15.24. A k-dimensional linear machine $\{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$ is said to be *μ-controllable* if for every pair of states S_i and S_j there is an input sequence of length *exactly* μ that takes the machine from state S_i to state S_j.

(a) Prove that a k-dimensional machine L is μ-controllable if and only if the rank of $k \times \mu l$ matrix

$$\mathbf{G}_\mu = [\mathbf{A}^{\mu-1}\mathbf{B} \quad \mathbf{A}^{\mu-2}\mathbf{B} \quad \cdots \quad \mathbf{A}\mathbf{B} \quad \mathbf{B}]$$

is k; i.e., there are k linearly independent columns in $\mathbf{G}_\mu$.

(b) Determine whether the following machine over $GF(2)$ is μ-controllable:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}.$$

Hint: Try the 3-controllable case first and show that $\mathbf{G}_3$ is singular.

Problem 15.25. For each machine in Table P15.25, determine whether it is linear and, if it is, show a linear realization.

Table P15.25

	NS, z	
PS	$x=0$	$x=1$
A	$A, 0$	$E, 1$
B	$E, 1$	$A, 0$
C	$F, 1$	$B, 0$
D	$B, 0$	$F, 1$
E	$C, 1$	$G, 0$
F	$G, 0$	$C, 1$
G	$H, 0$	$D, 1$
H	$D, 1$	$H, 0$

	NS				z_1z_2			
PS	00	01	11	10	00	01	11	10
A	E	F	A	B	10	11	00	01
B	G	H	C	D	11	10	01	00
C	B	A	F	E	01	00	11	10
D	D	C	H	G	00	01	10	11
E	B	A	F	E	11	10	01	00
F	D	C	H	G	10	11	00	01
G	E	F	A	B	00	01	10	11
H	G	H	C	D	01	00	11	10

Problem 15.26. Test the machine of Table P15.26 for linearity. In particular, determine whether the state transitions are linear and the outputs are linear.

Table P15.26

	NS, z	
PS	$x=0$	$x=1$
A	$A, 0$	$B, 0$
B	$C, 0$	$D, 0$
C	$A, 1$	$B, 1$
D	$C, 1$	$D, 0$

CHAPTER

16 Finite-state recognizers

In this chapter we consider the characterization of finite-state machines and the sets of sequences that they accept. We investigate a number of generalized forms of finite-state machines and prove that these forms are equivalent, with respect to the sets of sequences that they accept, to the basic deterministic finite-state model. In Sections 16.2 and 16.3 we study the properties of nondeterministic state diagrams, called transition graphs, which will prove to be a useful tool in the study of regular expressions. Procedures are developed whereby any transition graph can be converted into a deterministic state diagram.

Section 16.4 presents the language of regular expressions, which provides a precise characterization of the sets of sequences accepted by finite-state machines. In the following two sections we prove that any finite-state machine can be characterized by a regular expression and that every regular expression can be realized by a finite-state machine. Finally, in Section 16.7 we will be concerned with a generalized form of finite-state machines known as two-way machines.

16.1 Deterministic recognizers

So far, we have regarded a finite-state machine as a *transducer* that *transforms* input sequences into output sequences. In this chapter we shall view a machine as a *recognizer* that *classifies* input strings into two classes, those that it accepts and those that it rejects. The set consisting of all the strings that a given machine accepts is said to be *recognized* by that machine.

The finite-state model that we shall use is shown in Fig. 16.1, where a *finite-state control* is coupled through a *head* to a finite linear sequence of squares, each containing a single symbol of the alphabet. Such a sequence of squares is called an (*input*) *tape*. Initially, the finite-state control is in the starting state, and the head scans the leftmost symbol of the string that appears on the tape. The head then scans the tape from left to right. In what is termed

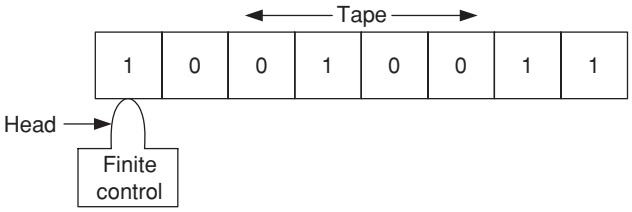
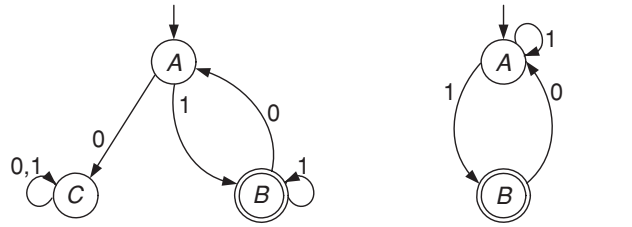


Fig. 16.1 A finite-state recognizer.



(*a*) Deterministic state diagram. (*b*) Transition graph.

Fig. 16.2 Two ways of describing a string.

a *cycle of computation*, the machine starts in some state S_i, reads the symbol currently scanned by the head, shifts one square to the right, and then enters the state S_j.

Clearly, the concept of a head reading from left to right the symbols contained in a linear tape is equivalent to a string of input symbols entering the machine at successive times. In fact, the finite-state control is a Moore finite-state machine.[1] States whose assigned output symbol is 1 are referred to as *accepting* (or *terminal*) *states* while states whose assigned output symbol is 0 are called *rejecting* (or *nonterminal*) *states*. A string (or a tape) is *accepted* by a machine if and only if the state that the machine enters after having read the rightmost tape symbol is an accepting state. Otherwise the string is rejected. The set of strings recognized by a machine thus consists of all the input strings that take the machine from its starting state to an accepting state.

The machine of Fig. 16.1 can be described by a state diagram in which the starting state is marked by an incoming short arrow and the accepting states are indicated by double circles. For example, the state diagram of Fig. 16.2*a* describes a machine that accepts a string if and only if the string begins and ends with a 1 and every 0 in the string is preceded and followed by at least a single 1. The machine consists of three states, of which A is the starting state and B is an accepting state. Note that in general a starting state may also be an accepting state. In such a case, the machine is said to accept the null string.

[1] By allowing the head to write on the tape, while restricting its motion to left-to-right, we can generalize the model to include Mealy machines.

16.2 Transition graphs

Because a state diagram describes a *deterministic* machine, the next-state transition must be determined *uniquely* by the present state and the currently scanned input symbol. No alternative behavior is allowed. Moreover, in a deterministic state diagram a transition must be specified for each input symbol. Consequently, a state diagram consists of a vertex for every state and a directed arc labeled α emanating from each vertex for every input symbol α. However, if our prime objective is to study and classify sets of sequences, some of these restrictions may be removed and different diagrams, called transition graphs, may prove more convenient.

Nondeterministic recognizers

A *transition graph* (or *transition system*) is a directed graph. It consists of a set of vertices labeled A, B, C, etc. and various directed arcs connecting them. At least one vertex is specified as a *starting vertex* and at least one is specified as an *accepting* (or *terminal*) *vertex*. The arcs are labeled with symbols from the (*input*) *alphabet* of the graph. If the graph contains an arc labeled α leading from vertex V_i to vertex V_j then V_j is said to be the α-*successor* of V_i. For a given input symbol α, a vertex may have one or more α-successors or none. Thus, for example, in the transition graph of Fig. 16.2*b*, vertex A has two 1-successors, namely A and B, but no 0-successor. A set of vertices S is said to be the α-successor of a set R if and only if every element of S is an α-successor of some element of R.

A sequence of directed arcs in a graph is referred to as a *path*. Every path is said to *describe* the string that consists of the symbols assigned to the arcs in the path. A string is accepted by a transition graph if it is described by at least one path that emanates from a starting vertex and terminates at an accepting vertex. Thus, for example, the string 1110 is accepted by the graph of Fig. 16.3, since it is described by a path that emanates from vertex A, passes through vertices B, D, and C, and terminates at vertex A. In the same manner, we find that the string 11011 is accepted by the graph, since it is described by a path that emanates from a starting vertex B, passes through D, C, B, D, and

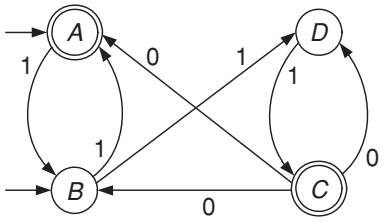


Fig. 16.3 A transition graph.

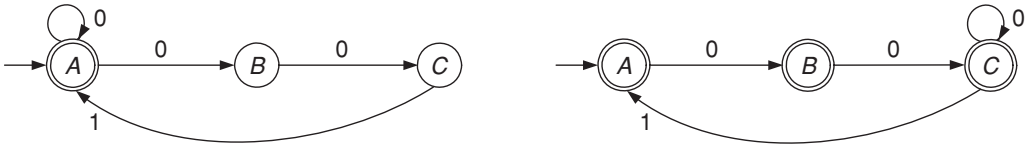


Fig. 16.4 Two equivalent transition graphs.

terminates at an accepting vertex C. However, the string 100, for example, is rejected since there is no path in the graph which describes it.

As in the case of state diagrams, the set of strings that are accepted by a transition graph is said to be *recognized* by the graph. For example, the transition graph of Fig. 16.2*b* recognizes the same set of strings as is recognized by the state diagram of Fig. 16.2*a*. If two or more graphs recognize the same set of strings then they are said to be *equivalent graphs*. Thus, the graphs in Fig. 16.4 are equivalent since each graph accepts a string if and only if each 1 in the string is preceded by at least two 0's.

Clearly, a state diagram is a special case of a transition graph and is, therefore, referred to as a *deterministic* (*transition*) *graph*. Other transition graphs are referred to as *nondeterministic* (*transition*) *graphs*. The two graphs in Fig. 16.2, for example, are equivalent although one is deterministic and the other is not. Because deterministic graphs describe the behavior of deterministic finite-state machines, we often regard nondeterministic graphs as describing the behavior of nondeterministic finite-state machines. It must, however, be emphasized that the notion of nondeterministic recognizers is useful for classifying sets of strings but should not be confused with the notion of realizable machines.

Graphs containing λ-transitions

Nondeterministic transition graphs can be generalized further by allowing transitions that are associated with a *null symbol* λ. Such transitions are referred to as λ-*transitions*, and they can occur when no input symbol is applied. When determining the string described by a path that contains arcs labeled λ, the λ-symbols are disregarded and deleted from the string.

The use of λ-transitions may sometimes simplify the transition graph by reducing the number of labeled arcs, as for the graph of Fig. 16.5*a*. This graph recognizes the set of strings that start with an even number of 1's, followed by an even number of 0's, and end up with substring 101. (Note that zero is considered as an even number.) Thus, for example, the strings 101, 11101, 110000101, and 00101 are accepted by the graph, while 110011101 and 0011101 are rejected.

It is a simple matter to convert a transition graph containing λ-transitions into an equivalent graph that contains no such transitions. A λ-transition from vertex V_1 to vertex V_2 of a given graph can always be replaced by a set of arcs emanating from V_1 and duplicating the transitions that emanate from V_2. In addition, if V_1 is a starting vertex then V_2 must also be made a starting vertex. If V_2 is an accepting vertex then V_1 must also be made an accepting

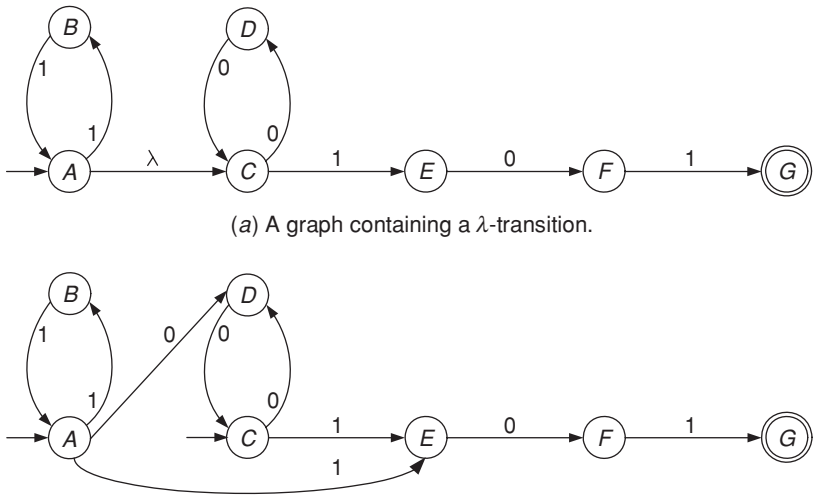


Fig. 16.5 Elimination of λ-transition.

(*a*) A graph containing a λ-transition.

(*b*) An equivalent graph without λ-transitions.

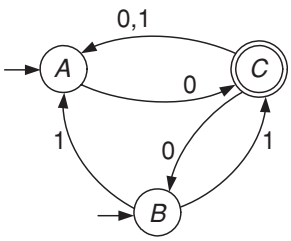
vertex. To remove the λ-transition from the graph of Fig. 16.5*a* it is necessary to duplicate the transitions from vertex C to vertices D and E by directing arcs, correspondingly labeled, from vertex A to vertices D and E. The equivalent graph that contains no λ-transition is shown in Fig. 16.5*b*.

16.3 Converting nondeterministic into deterministic graphs

A natural question, which now arises, is whether a nondeterministic graph can recognize sets of strings that cannot be recognized by a deterministic graph. At first, one might suspect that the added flexibility of nondeterministic graphs increases their computational capabilities. However, as we shall now show, *there exists an effective procedure for converting a nondeterministic transition graph into an equivalent deterministic transition graph.* This leads to the conclusion that nondeterministic graphs and deterministic graphs have identical computational capabilities.

Introductory example

Consider the nondeterministic transition graph of Fig. 16.6*a*. A tabular description of the graph, called a *transition table*, is shown in Fig. 16.6*b*, where the starting vertices are indicated by the small arrows next to rows A and B, and the accepting vertex is indicated by a circle around the row heading C. The table entry in row V_i, column α, consists of the α-successors of vertex V_i.



(*a*) Transition graph.

	0	1
→ *A*	*C*	—
→ *B*	—	*AC*
Ⓒ	*AB*	*A*

(*b*) Transition table.

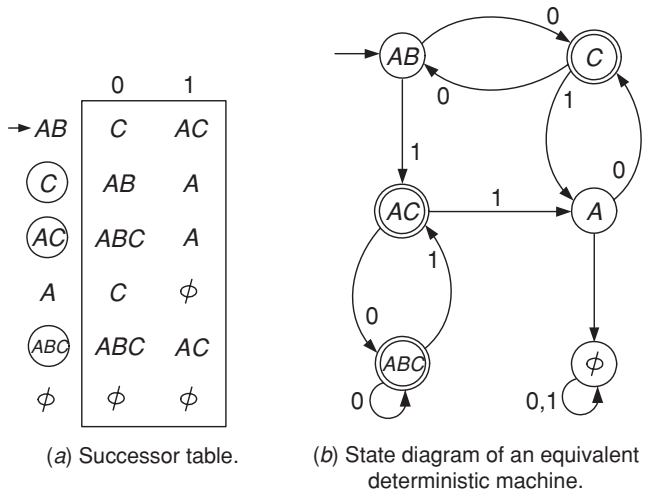
Fig. 16.6 A nondeterministic graph to be converted to a deterministic one.

Suppose now that we wish to determine whether a given string $w = a_1 a_2 \cdots a_k$ is accepted by the graph of Fig. 16.6*a*; that is, whether the graph contains a path that emanates from a starting vertex, terminates at an accepting vertex, and describes the string w. Since A and B are the starting vertices, any such path must include as its first arc an arc emanating from either A or B. Specifically, if the first symbol in w is a_1 then the first arc in the path can reach any vertex in the subset that consists of the a_1-successors of $\{A, B\}$. Using similar reasoning, we find that the ith arc in a path that describes w must lead to a vertex contained in the subset which consists of the $a_1 a_2 \cdots a_i$-successors of $\{A, B\}$. If the final subset of vertices reached by the path contains an accepting vertex then the string w is accepted; otherwise, it is rejected.

For example, any path that describes string 0010 must start with the arc leading from vertex A to vertex C. Also, since the 0-successors of C are A and B, one of these vertices must be encountered next in the path describing the given string. In the same manner, since $\{AC\}$ is the 1-successor of $\{AB\}$, we find that the third arc in the path leads to either of the vertices A or C. The fourth symbol might lead to one of the vertices A, B, or C and, since vertex C is an accepting vertex, the string is accepted. A similar argument shows, for example, that the string 1100 is rejected, since it might lead to either vertex A or vertex B and neither vertex is an accepting vertex.

The foregoing example suggests a procedure for determining whether a specified string is accepted by a given graph. The procedure involves tracing the various paths that describe the given string and determining the sets of vertices that can be reached from the starting vertices by applying the symbols of the string. The procedure can be facilitated and applied to arbitrary strings by the use of a *successor table*, which lists all the subsets of vertices that are reachable from the starting vertices. The successor table for the graph of Fig. 16.6 is shown in Fig. 16.7*a*. Its column headings are symbols of the alphabet. The first row heading is the set of starting vertices, while the remaining row headings are subsets of vertices reachable from starting vertices. The entry in row Q, column α, is determined from the transition table and consists of the α-successor of $\{Q\}$.

The first row heading in Fig. 16.7*a* is AB, since A and B are the starting vertices. The entries in row AB are the 0- and 1-successors of $\{AB\}$, namely



(*a*) Successor table. (*b*) State diagram of an equivalent deterministic machine.

Fig. 16.7 Deterministic form of the graph of Fig. 16.6.

$\{C\}$ and $\{AC\}$, respectively. The entries C and AC are now made row headings, their successors found, and so on. Since vertex A has no 1-successor, the 1-successor of row A must correspond to the set that contains no vertex of the transition graph. Such a set is referred to as the *empty*, or *null*, set and is denoted ϕ. Finally, the row headings of the rows C, AC, and ABC are circled to indicate that each of the sets $\{C\}$, $\{AC\}$, and $\{ABC\}$ contains the accepting vertex C of the original transition graph.

Proof of the conversion procedure

The graph in Fig. 16.7*b* is derived directly from the successor table. It is clearly a deterministic graph, since only one transition is allowed for each input symbol in its construction. To verify that this graph indeed accepts a given string if and only if that string is accepted by the corresponding nondeterministic graph, note that the last vertex of the deterministic graph reached by the string corresponds to the subset of vertices that can be reached by the same string in the nondeterministic graph. The string is accepted by the deterministic graph if and only if there is at least one path in the nondeterministic graph that results in the string being accepted, that is, if one vertex reachable by the string is an accepting vertex. The foregoing procedure, which is also known as *subset construction*, can be applied to any nondeterministic graph. Thus, we arrive at the following theorem.

Theorem 16.1 *Let S be a set of strings that can be recognized by a nondeterministic transition graph* G_n. *Then S can also be recognized by an equivalent deterministic graph* G_d. *Moreover, if* G_n *has p vertices then* G_d *will have at most* 2^p *vertices.*

Proof The existence of a deterministic graph G_{d} that is equivalent to the given nondeterministic graph G_{n} is guaranteed by the subset construction procedure developed above. If we denote the p vertices of G_{n} by $V_1, V_2, \ldots, V_p$, then, by subset construction, the equivalent deterministic graph may have at most 2^p vertices labeled as follows: ϕ, $V_1, V_2, \ldots, V_p$; V_1V_2, $V_1V_3, \ldots, V_2V_3, \ldots, V_{p-1}V_p$; $V_1V_2V_3, \ldots, V_{p-2}V_{p-1}V_p; \ldots;$ $V_1V_2 \ldots V_p$. ◇

Theorem 16.1 permits us to describe deterministic finite-state machines by means of nondeterministic transition graphs. Such descriptions will prove very convenient in the following discussion of regular expressions.

16.4 Regular expressions

In this chapter we are mainly concerned with the characterization of sets of strings recognized by finite automata. It is therefore appropriate to develop a compact language for describing such sets of strings. The language developed in this section is known as *type-3 language* or as the language of *regular expressions*.

Describing sets of strings

We shall first consider informally some sets recognized by simple graphs, leaving the formal presentation to subsequent sections. Consider the transition graph in Fig. 16.8*a*, which recognizes a set $\{101\}$ that contains just one string. We shall describe the set $\{101\}$ by the expression **101**.[2] Similarly, for an arbitrary alphabet $\{a, b\}$, the set $\{abba\}$ is described by the expression **abba**, and so on.

The graph in Fig. 16.8*b* recognizes the set of strings $\{01, 10\}$, that consists of two strings, 01 and 10. To represent such a set we employ the set union operation $+$, and express the set $\{01, 10\}$ as $\mathbf{01} + \mathbf{10}$. In the same manner, the set $\{abb, a, b, bba\}$ can be described by the expression $\mathbf{abb} + \mathbf{a} + \mathbf{b} + \mathbf{bba}$. Clearly, since the set union operation is commutative and associative, the union operation of expressions is also commutative and associative.

Next, consider the graph in Fig. 16.8*c*, which recognizes the set $\{0111, 1011\}$. This set can be described by the expression $\mathbf{0111} + \mathbf{1011}$. However, we observe that this graph recognizes precisely those strings that are recognized by the graph in Fig. 16.8*b and* which are followed immediately by the substring 11. In other words, the graph of Fig. 16.8*c* recognizes the set whose members are those strings formed by concatenating the strings in $\{01, 10\}$ and $\{11\}$. In general, the *concatenation* of two sets $\{P\}$ and $\{Q\}$ is the set

[2] In this chapter, boldface type is used to describe expressions.

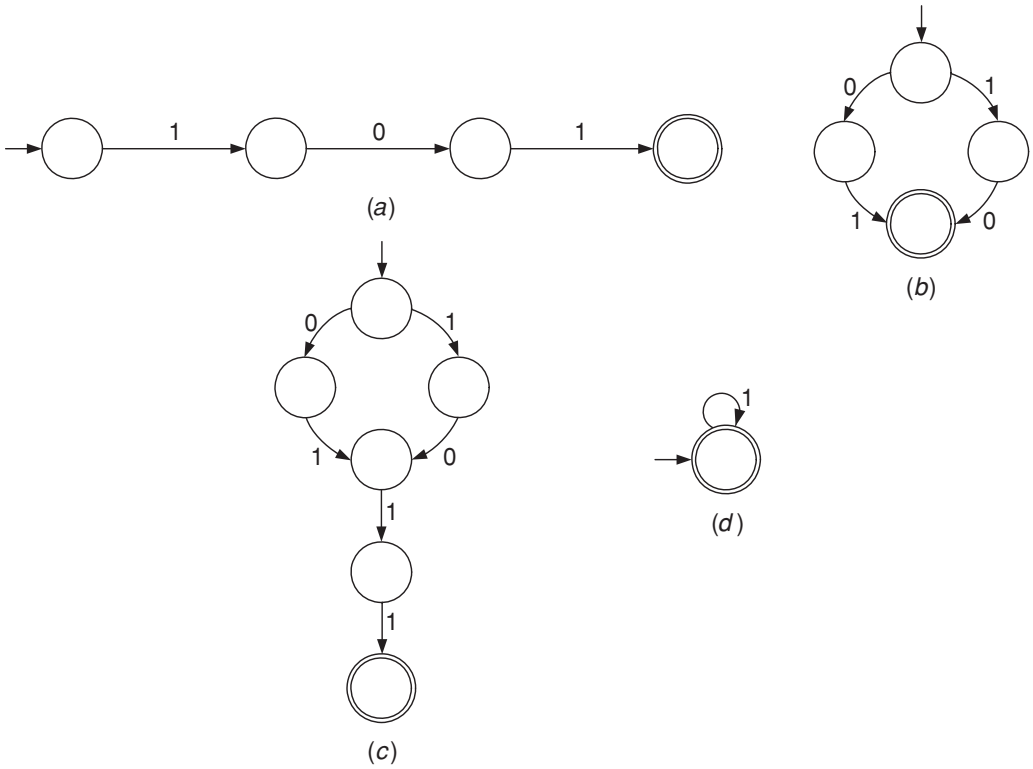


Fig. 16.8 Simple transition graphs.

consisting of strings formed by taking any string of $\{P\}$ and attaching to it any string of $\{Q\}$. The above set can thus be described by the *concatenation* of the two corresponding expressions $\mathbf{01} + \mathbf{10}$ and $\mathbf{11}$, i.e., $(\mathbf{01} + \mathbf{10})\mathbf{11}$. Clearly the concatenation operation is associative, that is, if **P**, **Q**, and **R** are expressions then $(\mathbf{PQ})\mathbf{R} = \mathbf{P}(\mathbf{QR})$, but it is not commutative, $\mathbf{PQ} \neq \mathbf{QP}$. To simplify the notation, we can omit the parentheses and write the product $(\mathbf{PQ})\mathbf{R}$ as **PQR**.

The graph in Fig. 16.8*d* recognizes the set of strings whose members consist of an arbitrary number (possibly zero) of 1's, i.e., $\{\lambda, 1, 11, 111, 1111, \ldots\}$. This set can be described by the infinite expression $\boldsymbol{\lambda} + \mathbf{1} + \mathbf{11} + \mathbf{111} + \mathbf{1111} + \cdots$ or, compactly, by $\mathbf{1}^*$, where

$$\mathbf{1}^* = \boldsymbol{\lambda} + \mathbf{1} + \mathbf{11} + \mathbf{111} + \mathbf{1111} + \cdots.$$

The symbol * is referred to as the *star* (or *closure*) *operation*. In general, $\mathbf{R}^*$ describes the set consisting of the null string λ and those strings that can be formed by concatenating a finite number of strings from $\{R\}$. For example, the expression $\mathbf{01}(\mathbf{01})^*$ describes the set consisting of those strings that can be formed by concatenating one or more 01 substrings, that is,

$$\mathbf{01}(\mathbf{01})^* = \mathbf{01} + \mathbf{0101} + \mathbf{010101} + \mathbf{01010101} + \cdots.$$

For convenience, **RR** may be abbreviated as $\mathbf{R}^2$, **RRR** as $\mathbf{R}^3$, etc. Thus,

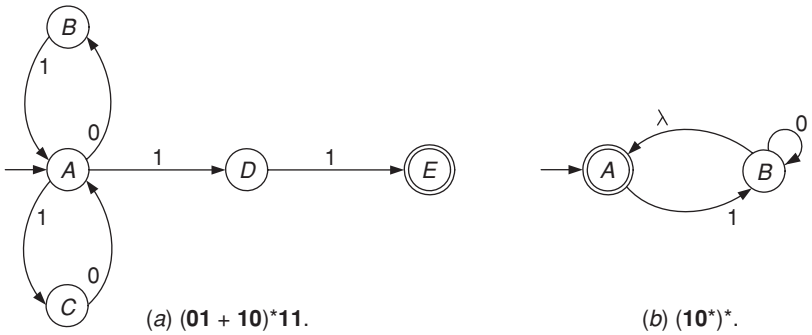
$$\mathbf{R}^* = \boldsymbol{\lambda} + \mathbf{R} + \mathbf{R}^2 + \mathbf{R}^3 + \cdots.$$

We are now able to describe some sets of strings on a given alphabet by means of the operations + , ·, *. For example, the set of strings on $\{0, 1\}$ beginning with a 0 and followed only by 1's can be described by $\mathbf{01}^*$ while the set of strings containing exactly two 1's can be described by $\mathbf{0}^*\mathbf{10}^*\mathbf{10}^*$. An important expression is $(\mathbf{0} + \mathbf{1})^*$, which describes the set containing all the strings that can be formed on the binary alphabet; that is,

$$(\mathbf{0} + \mathbf{1})^* = \boldsymbol{\lambda} + \mathbf{0} + \mathbf{1} + \mathbf{00} + \mathbf{01} + \mathbf{11} + \mathbf{10} + \mathbf{000} + \cdots.$$

Thus, for example, the set of strings that begin with the substring 11 is described by the expression $\mathbf{11}(\mathbf{0} + \mathbf{1})^*$.

Example The transition graph of Fig. 16.9*a* accepts those strings that can be formed by concatenating a finite number of 01 and 10 substrings followed by a 11. Accordingly, it can be described by the expression $(\mathbf{01} + \mathbf{10})^*\mathbf{11}$. In a similar manner, the reader can verify that the set of strings recognized by the graph of Fig. 16.9*b* can be described by $(\mathbf{10}^*)^*$.



(*a*) $(\mathbf{01} + \mathbf{10})^*\mathbf{11}$. (*b*) $(\mathbf{10}^*)^*$.

Fig. 16.9 Transition graphs and the sets of strings that they recognize.

We have thus shown that some sets of strings may be described by expressions formed of symbols from the alphabets of these sets and the operations union, concatenation, and star. We now formalize these ideas.

Definition and basic properties

Let $A = \{\alpha_1, \alpha_2, \ldots, \alpha_p\}$ be a finite *alphabet*; then the class of *regular expressions over alphabet A* is defined recursively as follows.

1. Any single *symbol* $\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \ldots, \boldsymbol{\alpha}_p$ is a regular expression, as are the *null string* $\boldsymbol{\lambda}$ and the *empty set* $\boldsymbol{\phi}$.

Fig. 16.10 Recognizers for λ and $\boldsymbol{\phi}$.

(*a*) A graph accepting $\boldsymbol{\lambda}$. (*b*) A graph accepting $\boldsymbol{\phi}$.

2. If **P** and **Q** are regular expressions then so is their *concatenation* **PQ** and their *union* $\mathbf{P} + \mathbf{Q}$. If **P** is a regular expression then so is its *closure* $\mathbf{P}^*$.
3. No other expressions are regular unless they can be generated in a *finite* number of applications of the above rules.

By convention, the precedence of the operations in decreasing order is *, ·, +.

At this point, it is appropriate to consider the significance of the expressions $\boldsymbol{\lambda}$ and $\boldsymbol{\phi}$. The expression $\boldsymbol{\lambda}$ describes the set that consists of just the null string. It can be recognized, for example, by the graph of Fig. 16.10*a*. Expression $\boldsymbol{\phi}$, however, describes the set that has no strings at all. In other words, $\boldsymbol{\phi}$ describes the set recognized by a graph that accepts no strings, such as the graph shown in Fig. 16.10*b*. The reader may verify that each of the following identities, which involve the expressions $\boldsymbol{\phi}$ and $\boldsymbol{\lambda}$, exhibits different ways of describing the *same* sets of strings:

$$\boldsymbol{\phi} + \mathbf{R} = \mathbf{R}, \tag{16.1}$$

$$\boldsymbol{\phi}\mathbf{R} = \mathbf{R}\boldsymbol{\phi} = \boldsymbol{\phi}, \tag{16.2}$$

$$\mathbf{R}\boldsymbol{\lambda} = \boldsymbol{\lambda}\mathbf{R} = \mathbf{R}, \tag{16.3}$$

$$\boldsymbol{\lambda}^* = \boldsymbol{\lambda}, \tag{16.4}$$

$$\boldsymbol{\phi}^* = \boldsymbol{\lambda}. \tag{16.5}$$

A set of strings that can be described by a regular expression is called a *regular set*. Not every set of strings is regular. For example, the set over the alphabet $\{0, 1\}$ that consists of k 0's (for all k), followed by a 1, followed in turn by k 0's is not regular, as will be proved later. This set can be described by the expression $\mathbf{010} + \mathbf{00100} + \mathbf{0001000} + \cdots + \mathbf{0}^k\mathbf{10}^k + \cdots$. However, such a description involves an infinite number of applications of the union operation. Consequently, it is not a regular expression. There are, however, certain infinite sums that are regular. For example, the set that consists of alternating 0's and 1's, starting and ending with a 1, i.e., {1, 101, 10101, 1010101, ...}, can be described by the expression $\mathbf{1} + \mathbf{101} + \mathbf{10101} + \cdots$, or $\mathbf{1}(\mathbf{01})^*$, which is clearly regular.

Manipulating regular expressions

A regular set may be described by more than one regular expression. For example, the above set of alternating 0's and 1's can be described by the expression $\mathbf{1}(\mathbf{01})^*$, as well as by $(\mathbf{10})^*\mathbf{1}$. Two expressions that describe the same set of strings are said to be *equivalent*. Unfortunately, no straightforward methods are

available to determine whether two given expressions are equivalent. In certain cases, however, a regular expression can be converted into another equivalent expression by the use of simple identities. Some of these identities (whose proofs are left to the reader as an exercise) are listed as follows.

Let **P**, **Q**, and **R** be regular expressions; then

$$\mathbf{R} + \mathbf{R} = \mathbf{R}, \tag{16.6}$$

$$\mathbf{PQ} + \mathbf{PR} = \mathbf{P}(\mathbf{Q} + \mathbf{R}), \quad \mathbf{PQ} + \mathbf{RQ} = (\mathbf{P} + \mathbf{R})\mathbf{Q}, \tag{16.7}$$

$$\mathbf{R}^*\mathbf{R}^* = \mathbf{R}^*, \tag{16.8}$$

$$\mathbf{RR}^* = \mathbf{R}^*\mathbf{R}, \tag{16.9}$$

$$(\mathbf{R}^*)^* = \mathbf{R}^*, \tag{16.10}$$

$$\boldsymbol{\lambda} + \mathbf{RR}^* = \mathbf{R}^*, \tag{16.11}$$

$$(\mathbf{PQ})^*\mathbf{P} = \mathbf{P}(\mathbf{QP})^*. \tag{16.12}$$

To prove the last identity, note that each of the expressions $(\mathbf{PQ})^*\mathbf{P}$ and $\mathbf{P}(\mathbf{QP})^*$ can be written in the form $\mathbf{P} + \mathbf{PQP} + \mathbf{PQPQP} + \cdots$.

The set described by the expression $(\mathbf{P} + \mathbf{Q})^*$ consists of all the strings that can be formed by concatenating **P**'s and **Q**'s, including the null string $\boldsymbol{\lambda}$. It is easy to verify that the expression $(\mathbf{P}^* + \mathbf{Q}^*)^*$ describes the same set of strings, as does the expression $(\mathbf{P}^*\mathbf{Q}^*)^*$. Thus, we find that

$$(\mathbf{P} + \mathbf{Q})^* = (\mathbf{P}^*\mathbf{Q}^*)^* = (\mathbf{P}^* + \mathbf{Q}^*)^*. \tag{16.13}$$

However, note that $(\mathbf{P} + \mathbf{Q})^* \neq \mathbf{P}^* + \mathbf{Q}^*$.

The following identity will be proved in Section 16.5:

$$(\mathbf{P} + \mathbf{Q})^* = \mathbf{P}^*(\mathbf{QP}^*)^* = (\mathbf{P}^*\mathbf{Q})^*\mathbf{P}^*. \tag{16.14}$$

This identity leads in turn to

$$\boldsymbol{\lambda} + (\mathbf{P} + \mathbf{Q})^*\mathbf{Q} = (\mathbf{P}^*\mathbf{Q})^*. \tag{16.15}$$

Indeed, by Eqs. (16.11) and (16.14),

$$\begin{aligned}(\mathbf{P}^*\mathbf{Q})^* &= \boldsymbol{\lambda} + (\mathbf{P}^*\mathbf{Q})^*\mathbf{P}^*\mathbf{Q} \\ &= \boldsymbol{\lambda} + (\mathbf{P} + \mathbf{Q})^*\mathbf{Q}.\end{aligned}$$

The preceding identities can sometimes be used to simplify regular expressions or demonstrate their equivalence, as illustrated in the following examples.

Example Prove that the set of strings in which every 0 is immediately followed by at least two 1's can be described by both $\mathbf{R}_1$ and $\mathbf{R}_2$, where

$$\mathbf{R}_1 = \boldsymbol{\lambda} + \mathbf{1}^*(\mathbf{011})^*(\mathbf{1}^*(\mathbf{011})^*)^*,$$
$$\mathbf{R}_2 = (\mathbf{1} + \mathbf{011})^*.$$

We proceed as follows.

$$\begin{aligned}\mathbf{R}_1 &= \boldsymbol{\lambda} + \mathbf{1}^*(\mathbf{011})^*(\mathbf{1}^*(\mathbf{011})^*)^* \qquad \text{(by (16.11))}\\ &= (\mathbf{1}^*(\mathbf{011})^*)^* \qquad \text{(by (16.13))}\\ &= (\mathbf{1}+\mathbf{011})^* = \mathbf{R}_2.\end{aligned}$$

The reader can verify that R_2 indeed describes the set in question.

Example Prove the identity

$$(\mathbf{1}+\mathbf{00}^*\mathbf{1}) + (\mathbf{1}+\mathbf{00}^*\mathbf{1})(\mathbf{0}+\mathbf{10}^*\mathbf{1})^*(\mathbf{0}+\mathbf{10}^*\mathbf{1}) = \mathbf{0}^*\mathbf{1}(\mathbf{0}+\mathbf{10}^*\mathbf{1})^*.$$

Consider the left-hand side:

$$\begin{aligned}&(\mathbf{1}+\mathbf{00}^*\mathbf{1}) + (\mathbf{1}+\mathbf{00}^*\mathbf{1})(\mathbf{0}+\mathbf{10}^*\mathbf{1})^*(\mathbf{0}+\mathbf{10}^*\mathbf{1})\\ &\quad= (\mathbf{1}+\mathbf{00}^*\mathbf{1})[\boldsymbol{\lambda} + (\mathbf{0}+\mathbf{10}^*\mathbf{1})^*(\mathbf{0}+\mathbf{10}^*\mathbf{1})]\\ &\quad= [(\boldsymbol{\lambda}+\mathbf{00}^*)\mathbf{1}][\boldsymbol{\lambda} + (\mathbf{0}+\mathbf{10}^*\mathbf{1})^*(\mathbf{0}+\mathbf{10}^*\mathbf{1})] \quad \text{(by (16.11))}\\ &\quad= \mathbf{0}^*\mathbf{1}(\mathbf{0}+\mathbf{10}^*\mathbf{1})^*.\end{aligned}$$

In many situations, however, algebraic manipulations of regular expressions are extremely involved and thus are not a suitable tool for determining the equivalence of two regular expressions. As we shall see in the next section, perhaps the best approach is to convert the expressions in question into their equivalent state diagrams and to test the diagrams for equivalence by the techniques of Chapter 10. Other procedures for establishing the equivalence of regular expressions can be found in [3].

16.5 Transition graphs recognizing regular sets

We have already seen in several examples that transition graphs are capable of recognizing regular sets. We wish to show now that to every regular set there corresponds a transition graph (and hence a deterministic finite-state machine) that recognizes that set of strings.

Constructing the transition graphs

We now prove the following theorem.

Theorem 16.2 *Every regular expression* **R** *can be recognized by a transition graph.*

Proof We shall prove the theorem by constructing the required transition graph. The construction procedure is inductive on the total number of characters in **R**, where by a *character* we refer to an appearance of any of the expressions

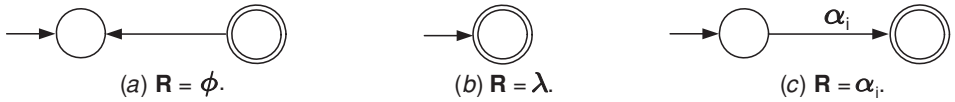


Fig. 16.11 Transition graphs recognizing elementary regular sets.

$\alpha_1, \alpha_2, \ldots, \alpha_p, \lambda, \phi$ or the star operation * in $\mathbf{R}$. For example, the number of characters in $\mathbf{R} = \lambda + (\mathbf{1}^*\mathbf{0})^*\mathbf{1}^*$ is seven.

Basis Let the number of characters in $\mathbf{R}$ be one. Then $\mathbf{R}$ must be either ϕ, λ, or a symbol, say α_i, from the alphabet. The graphs in Fig. 16.11 recognize these regular sets.[3]

Induction step Assume the theorem is true for expressions with n or fewer characters. We now show that it must also be true for any expression $\mathbf{R}$ having $n + 1$ characters. The expression $\mathbf{R}$ must be in one of the following three forms:

1. $\mathbf{R} = \mathbf{P} + \mathbf{Q}$,
2. $\mathbf{R} = \mathbf{PQ}$,
3. $\mathbf{R} = \mathbf{P}^*$,

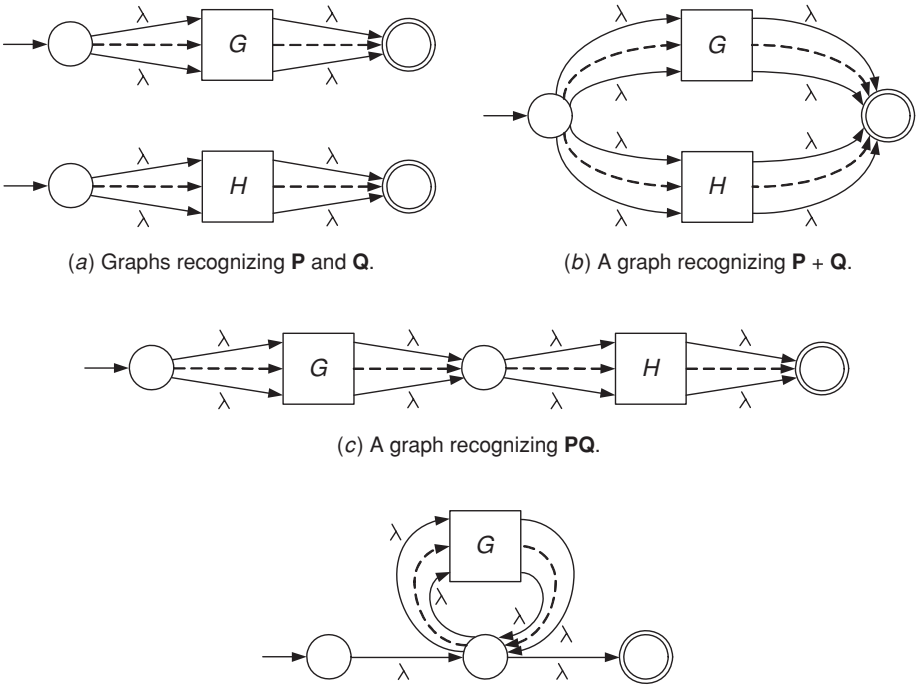
where $\mathbf{P}$ and $\mathbf{Q}$ are each expressions having n or fewer characters. According to the induction hypothesis, the sets $\mathbf{P}$ and $\mathbf{Q}$ can be recognized by transition graphs, which we shall denote G and H, respectively, as shown in Fig. 16.12*a*. (Note that each graph in Fig. 16.12 contains just one starting and one accepting vertex.)

The set described by $\mathbf{P} + \mathbf{Q}$ can be recognized by a transition graph composed of G and H, as shown in Fig. 16.12*b*. The set described by $\mathbf{PQ}$ can be recognized by a transition graph constructed in the following manner. Coalesce the accepting vertex of G with the starting vertex of H and regard the combined vertex as one that is neither starting nor accepting. The resulting graph is shown in Fig. 16.12*c*. The starting vertices of this graph are the starting vertices of G, while the accepting vertices are those of H. Clearly, this graph will accept a string if and only if that string belongs to $\mathbf{R} = \mathbf{PQ}$. Finally, to recognize the set $\mathbf{P}^*$, construct the graph of Fig. 16.12*d*. The graphs in Fig. 16.12, which are composed of G and H, are referred to as *composite graphs*.

Since every regular set can be described by an expression obtained by a finite number of applications of operations $+, \cdot, *$ on an alphabet $\{\alpha_1, \alpha_2, \ldots, \alpha_p\}$, ϕ and λ, the theorem is proved. ◇

The foregoing proof makes it possible to state an upper bound on the number of vertices in a graph that recognizes a given regular expression $\mathbf{R}$. Every graph clearly contains one starting and one accepting vertex. Subexpressions connected by the $+$ operation yield a composite graph that has as many vertices as the sum of vertices in the graphs that recognize individual subexpressions.

[3] Although there is a distinction between regular expressions and the sets that they describe, it is customary to speak of the regular set $\mathbf{R}$ as the set that can be described by the expression $\mathbf{R}$.



(*a*) Graphs recognizing **P** and **Q**.

(*b*) A graph recognizing **P** + **Q**.

(*c*) A graph recognizing **PQ**.

(*d*) A graph recognizing **P***.

Fig. 16.12 Construction of composite graphs.

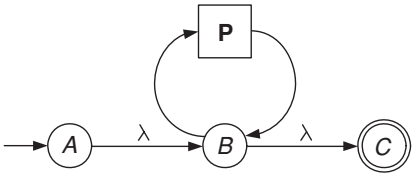
Two subexpressions connected by the concatenation operation add a new vertex to the composite graph, and similarly for the closure operation *. By induction on the number of vertices, we find that the number of vertices v in a graph that recognizes the given expression **R** need not exceed

$$v = 2 + \text{number of concatenations} + \text{number of stars}.$$

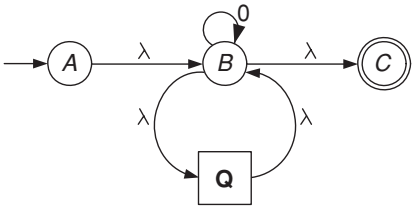
Theorem 16.2 provides us with a procedure for constructing a transition graph that recognizes a given regular expression **R**. Converting the graph to a deterministic form yields a state diagram of a finite-state machine that recognizes the set **R**.

Example Consider the regular expression $\mathbf{R} = (\mathbf{0} + \mathbf{1}(\mathbf{01})^*)^*$. Since it is of the form $\mathbf{P}^*$, where $\mathbf{P} = \mathbf{0} + \mathbf{1}(\mathbf{01})^*$, it is recognized by the graph of Fig. 16.13*a*. We now observe that $\mathbf{P} = \mathbf{0} + \mathbf{Q}$, where $\mathbf{Q} = \mathbf{1}(\mathbf{01})^*$, and the resulting graph is shown in Fig. 16.13*b*. The subexpression **Q** can be decomposed into $\mathbf{Q} = \mathbf{ST}$, where $\mathbf{S} = \mathbf{1}$ and $\mathbf{T} = (\mathbf{01})^*$. This yields the graph of Fig. 16.13*c*. The process is continued in a similar manner until each subexpression consists of only a single symbol. The final transition graph that

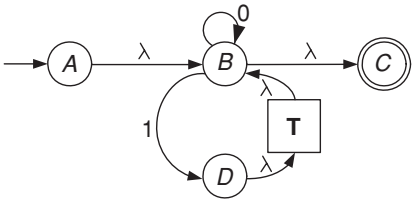
recognizes **R** is shown in Fig. 16.13*d*. Note that the number of vertices in the graph is six, in agreement with the value of v derived above.



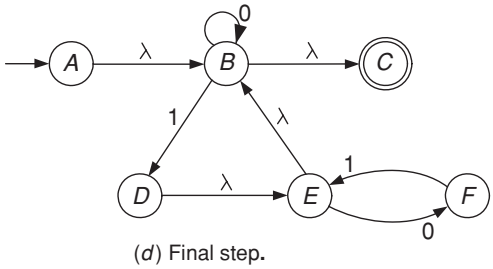
(*a*) **R = P*; P = 0 + 1(01)*.**



(*b*) **P = 0 + Q; Q = 1(01)*.**



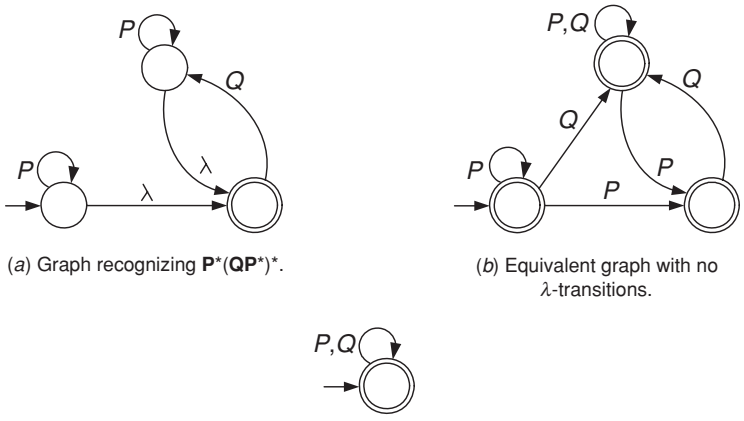
(*c*) **Q = 1T; T = (01)*.**



(*d*) Final step.

Fig. 16.13 Construction of a transition graph recognizing $\mathbf{R} = (\mathbf{0} + \mathbf{1}(\mathbf{01})^*)^*$.

We can now prove the first identity in Eq. (16.14) by demonstrating that the expressions $(\mathbf{P} + \mathbf{Q})^*$ and $\mathbf{P}^*(\mathbf{QP}^*)^*$ can be recognized by equivalent transition graphs. The graph in Fig. 16.14*a* recognizes the set described by $\mathbf{P}^*(\mathbf{QP}^*)^*$. Removal of the λ-transitions results in the graph of Fig. 16.14*b*, which can be converted to the deterministic graph of Fig. 16.14*c*. Clearly this graph recognizes set $(\mathbf{P} + \mathbf{Q})^*$, and thus the two expressions are equivalent. By Eq. (16.12), we obtain $\mathbf{P}^*(\mathbf{QP}^*)^* = (\mathbf{P}^*\mathbf{Q})^*\mathbf{P}^*$, which proves the second identity.



(*a*) Graph recognizing $\mathbf{P}^*(\mathbf{QP}^*)^*$.

(*b*) Equivalent graph with no λ-transitions.

(*c*) Equivalent deterministic graph recognizing $(\mathbf{P}+\mathbf{Q})^*$.

Fig. 16.14 Illustration of the proof that $\mathbf{P}^*(\mathbf{QP}^*)^* = (\mathbf{P}+\mathbf{Q})^*$.

Informal techniques

In practice, in many cases it is possible to construct transition graphs from their corresponding regular expressions in a straightforward manner, without resorting to the above induction procedure.

Example Construct a graph that recognizes the regular set

$$\mathbf{P} = (\mathbf{01} + (\mathbf{11} + \mathbf{0})\mathbf{1}^*\mathbf{0})^*\mathbf{11}.$$

As an introduction, we shall construct a graph that recognizes the subexpression $\mathbf{Q} = (\mathbf{11} + \mathbf{0})\mathbf{1}^*\mathbf{0}$. Every string in $\mathbf{Q}$ starts with one of the substrings 11 and 0, followed by an arbitrary number of 1's, and ends with a 0. The graph of Fig. 16.15 clearly recognizes just this set of strings. The subexpressions **11** and **0** are represented by parallel paths between the vertices A and C, while 1^* corresponds to a self-loop around vertex C. To ensure that a string is accepted only if it ends with a 0, an arc labeled 0 leads from vertex C to accepting vertex D.

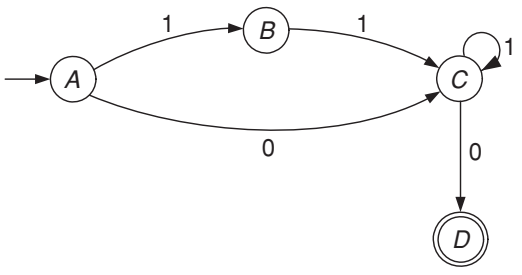


Fig. 16.15 A graph recognizing $\mathbf{Q} = (\mathbf{11} + \mathbf{0})\mathbf{1}^*\mathbf{0}$.

Now consider expression **P**. The graph that recognizes **P** is constructed in such a way that paths are provided for strings from the sets **01** and $(\mathbf{11} + \mathbf{0})\mathbf{1}^*\mathbf{0}$, followed by a string from the set **11**. One such possible graph is shown in Fig. 16.16.

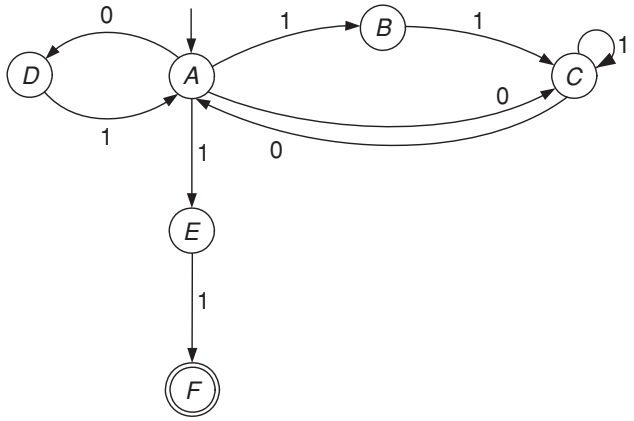


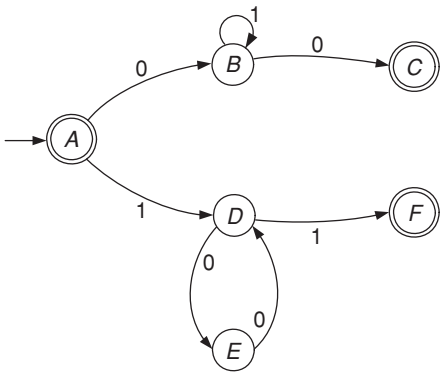
Fig. 16.16 A graph recognizing $\mathbf{P} = (\mathbf{01} + (\mathbf{11} + \mathbf{0})\mathbf{1}^*\mathbf{0})^*\mathbf{11}$.

In a number of cases it is convenient to use λ-transitions to preserve the order in which substrings appear. As an example, consider the expression $\mathbf{R} = (\mathbf{11})^*(\mathbf{00})^*\mathbf{101}$. In this expression, substrings from $(\mathbf{00})^*$ must follow substrings from $(\mathbf{11})^*$. One way of ensuring that this order is preserved is by using a λ-transition, as shown in Fig. 16.5*a*. This graph accepts only those strings that start with a substring from $(\mathbf{11})^*$, continue with a substring from $(\mathbf{00})^*$, and end with the substring 101.

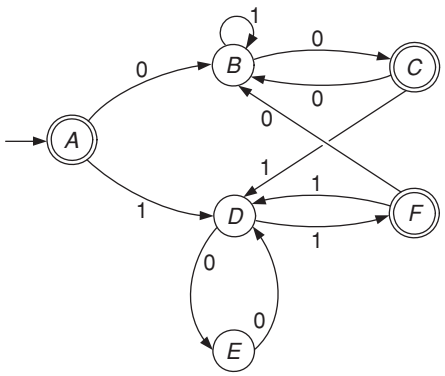
Example Construct a transition graph that recognizes the set

$$\mathbf{R} = (\mathbf{1}(\mathbf{00})^*\mathbf{1} + \mathbf{01}^*\mathbf{0})^*.$$

We begin by setting up paths for the subexpressions $\mathbf{1}(\mathbf{00})^*\mathbf{1}$ and $\mathbf{01}^*\mathbf{0}$, as shown in Fig. 16.17*a*. Vertex A is the starting vertex, while A, C, and F are accepting vertices. To complete the graph, an arc labeled α_i is drawn from vertex V_j to vertex V_k if and only if a sequence leading from the starting vertex to V_j that is followed by α_i and then by a sequence that emanates from V_k to an accepting vertex is an acceptable sequence. Accordingly, for example, an arc labeled 0 is drawn from F to B since 1100 is an acceptable sequence. The graph is completed in a similar manner, as shown in Fig. 16.17*b*.



(*a*) Partial graph.



(*b*) Complete graph.

Fig. 16.17 Transition graph recognizing $\mathbf{R} = (\mathbf{1}(\mathbf{00})^*\mathbf{1} + \mathbf{01}^*\mathbf{0})^*$.

In conclusion, we have established that every regular set can be recognized by a finite-state machine. Moreover, there is a routine procedure for determining the machine that recognizes a given regular set. This procedure involves the use of nondeterministic transition graphs, which can later be converted into the equivalent deterministic graphs. Other methods, however, are available [6] that provide a state-diagram description of the machine directly, without the need to resort to nondeterministic graphs.

16.6 Regular sets corresponding to transition graphs

We now consider the problem of deriving regular expressions that describe specified transition graphs. Specifically, we shall show that the set of strings that can be recognized by a transition graph (and hence a finite-state machine) is a regular set.

Proof of uniqueness

Before proceeding with our main topic, we shall establish the following theorem.

Theorem 16.3 *Let* $\mathbf{Q}$, $\mathbf{P}$, *and* $\mathbf{R}$ *be regular expressions on a finite alphabet. Then, if* $\mathbf{P}$ *does not contain* $\boldsymbol{\lambda}$, *the equation*

$$\mathbf{R} = \mathbf{Q} + \mathbf{RP} \tag{16.16}$$

has a unique solution given by

$$\mathbf{R} = \mathbf{QP}^*. \tag{16.17}$$

Proof Clearly, $\mathbf{R} = \mathbf{QP}^*$ is a solution to the equation $\mathbf{R} = \mathbf{Q} + \mathbf{RP}$, since (by substitution and Eq. (16.11))

$$\mathbf{R} = \mathbf{Q} + \mathbf{RP} = \mathbf{Q} + \mathbf{QP}^*\mathbf{P} = \mathbf{Q}(\boldsymbol{\lambda} + \mathbf{P}^*\mathbf{P}) = \mathbf{QP}^*.$$

To prove uniqueness, make the expansion

$$\begin{aligned}
\mathbf{R} &= \mathbf{Q} + \mathbf{RP} \\
&= \mathbf{Q} + (\mathbf{Q} + \mathbf{RP})\mathbf{P} = \mathbf{Q} + \mathbf{QP} + \mathbf{RP}^2 \\
&= \mathbf{Q} + \mathbf{QP} + (\mathbf{Q} + \mathbf{RP})\mathbf{P}^2 = \mathbf{Q} + \mathbf{QP} + \mathbf{QP}^2 + \mathbf{RP}^3 \\
&\;\;\vdots \\
&= \mathbf{Q}(\boldsymbol{\lambda} + \mathbf{P} + \mathbf{P}^2 + \cdots + \mathbf{P}^{i-1} + \mathbf{P}^i) + \mathbf{RP}^{i+1},
\end{aligned} \tag{16.18}$$

where i is any arbitrary integer. Choose some string w in $\mathbf{R}$, suppose that the length of w is k, and then substitute $i = k$ into Eq. (16.18):

$$\mathbf{R} = \mathbf{Q}(\boldsymbol{\lambda} + \mathbf{P} + \mathbf{P}^2 + \cdots + \mathbf{P}^k) + \mathbf{RP}^{k+1}.$$

Since $\mathbf{P}$ does not contain $\boldsymbol{\lambda}$, the length of the shortest string in the set $\mathbf{RP}^{k+1}$ is at least $k + 1$. Consequently, w is not contained in $\mathbf{RP}^{k+1}$, but is contained in $\mathbf{Q}(\boldsymbol{\lambda} + \mathbf{P} + \mathbf{P}^2 + \cdots + \mathbf{P}^k)$. However, since $\mathbf{Q}(\boldsymbol{\lambda} + \mathbf{P} + \mathbf{P}^2 + \cdots + \mathbf{P}^k)$ is contained in $\mathbf{QP}^*$, w is contained in $\mathbf{QP}^*$.

To prove the converse, suppose that w is a string in $\mathbf{QP}^*$. Then there exists some integer k such that w is in $\mathbf{QP}^k$. This, in turn, implies that w is contained in $\mathbf{Q}(\boldsymbol{\lambda} + \mathbf{P} + \mathbf{P}^2 + \cdots + \mathbf{P}^k)$ and hence in $\mathbf{R} = \mathbf{Q} + \mathbf{RP}$. ◇

In an analogous manner, we can show that if $\mathbf{P}$ does not contain $\boldsymbol{\lambda}$ then $\mathbf{R} = \mathbf{P}^*\mathbf{Q}$ is the unique solution to the equation $\mathbf{R} = \mathbf{Q} + \mathbf{PR}$. Note that if $\mathbf{P}$ contains $\boldsymbol{\lambda}$, the solution of Eq. (16.16) is not unique. If $\mathbf{P} = \boldsymbol{\phi}$ then $\mathbf{R} = \mathbf{Q}$.

*Systems of equations

Consider the transition graph of Fig. 16.18, whose starting vertex is A and accepting vertex C. The set of strings recognized by this graph consists of all the strings that can be described by paths emanating from vertex A and

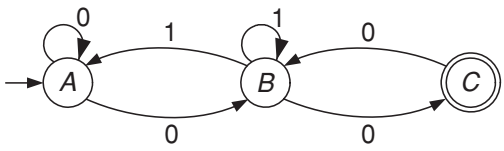


Fig. 16.18 A transition graph to be analyzed.

terminating at vertex C. However, since vertex C can be reached only through vertex B, each of these strings must end with a 0 and have as prefix a string leading from A to B. Let us denote the set of strings leading from A to B by $\mathbf{B}$ and the set of strings that take the graph from A to C by $\mathbf{C}$. Set $\mathbf{C}$ can then be expressed as $\mathbf{C} = \mathbf{B0}$.

Next consider set $\mathbf{A}$, which consists of exactly those strings that take the graph from vertex A to itself. Vertex A can be reached from B with a 1, from A with a 0, and with the null string λ. Thus, $\mathbf{A}$ can be expressed as $\mathbf{A} = \boldsymbol{\lambda} + \mathbf{A0} + \mathbf{B1}$. Finally, vertex B can be reached from A with a 0, from B with a 1, and from C with a 0. As a result, we obtain the equation $\mathbf{B} = \mathbf{A0} + \mathbf{B1} + \mathbf{C0}$.

The foregoing analysis yields a system of three simultaneous equations which characterize the sets of strings that take the graph from its starting vertex to each of its vertices. In Theorem 16.4 we shall prove that each of these sets of strings is regular, i.e.,

$$\mathbf{A} = \boldsymbol{\lambda} + \mathbf{A0} + \mathbf{B1}, \tag{16.19}$$

$$\mathbf{B} = \mathbf{A0} + \mathbf{B1} + \mathbf{C0}, \tag{16.20}$$

$$\mathbf{C} = \mathbf{B0}. \tag{16.21}$$

These equations can now be solved for the variables $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. Substituting Eq. (16.21) for $\mathbf{C}$ into Eq. (16.20) yields

$$\mathbf{B} = \mathbf{A0} + \mathbf{B1} + \mathbf{B00} = \mathbf{A0} + \mathbf{B}(\mathbf{1} + \mathbf{00}). \tag{16.22}$$

Equation (16.22) is now of the form of Eq. (16.16),

$$\mathbf{R} = \mathbf{Q} + \mathbf{RP},$$

and its solution is given by Eq. (16.17), i.e.,

$$\mathbf{R} = \mathbf{QP}^*.$$

Applying Eq. (16.17) to Eq. (16.22), we obtain

$$\mathbf{B} = \mathbf{A0}(\mathbf{1} + \mathbf{00})^*. \tag{16.23}$$

Now $\mathbf{B}$ can be substituted into Eq. (16.19) to give

$$\mathbf{A} = \boldsymbol{\lambda} + \mathbf{A0} + \mathbf{A0}(\mathbf{1} + \mathbf{00})^*\mathbf{1} = \boldsymbol{\lambda} + \mathbf{A}(\mathbf{0} + \mathbf{0}(\mathbf{1} + \mathbf{00})^*\mathbf{1}). \tag{16.24}$$

Equation (16.24) is again of the general form of Eq. (16.16) and, thus, has the solution

$$\mathbf{A} = \boldsymbol{\lambda}(\mathbf{0} + \mathbf{0}(\mathbf{1} + \mathbf{00})^*\mathbf{1})^* = (\mathbf{0} + \mathbf{0}(\mathbf{1} + \mathbf{00})^*\mathbf{1})^*. \tag{16.25}$$

Since the set recognized by the graph is given by **C**, we want to find a solution for this variable. Substituting Eq. (16.25) for **A** into Eq. (16.23), we obtain a solution for **B** that, in turn, may be substituted into Eq. (16.21) to yield the solution for **C**, i.e.,

$$\mathbf{B} = (\mathbf{0} + \mathbf{0}(\mathbf{1} + \mathbf{00})^*\mathbf{1})^*\mathbf{0}(\mathbf{1} + \mathbf{00})^*, \tag{16.26}$$

$$\mathbf{C} = (\mathbf{0} + \mathbf{0}(\mathbf{1} + \mathbf{00})^*\mathbf{1})^*\mathbf{0}(\mathbf{1} + \mathbf{00})^*\mathbf{0}. \tag{16.27}$$

The above procedure can now be applied to find a system of simultaneous equations for any transition graph that contains no λ-transitions and has a single starting vertex. (Recall that every transition graph can be converted to an equivalent graph with no λ-transitions and just one starting vertex.) Suppose that V_1 is the starting vertex in a graph containing n vertices, $V_1, V_2, \ldots, V_n$. Let $\mathbf{V}_i$ denote the set of strings that take the graph from V_1 to V_i, and let $\boldsymbol{\alpha}_{ij}$ denote the set of strings that take the graph from vertex V_i to vertex V_j without going through any other vertex; $\boldsymbol{\alpha}_{ij} = \boldsymbol{\phi}$ if no direct transition exists from V_i to V_j. Then we arrive at the following equations:

$$\begin{aligned}
\mathbf{V}_1 &= \mathbf{V}_1\boldsymbol{\alpha}_{11} + \mathbf{V}_2\boldsymbol{\alpha}_{21} + \cdots + \mathbf{V}_n\boldsymbol{\alpha}_{n1} + \boldsymbol{\lambda}, \\
\mathbf{V}_2 &= \mathbf{V}_1\boldsymbol{\alpha}_{12} + \mathbf{V}_2\boldsymbol{\alpha}_{22} + \cdots + \mathbf{V}_n\boldsymbol{\alpha}_{n2}, \\
&\vdots \\
\mathbf{V}_n &= \mathbf{V}_1\boldsymbol{\alpha}_{1n} + \mathbf{V}_2\boldsymbol{\alpha}_{2n} + \cdots + \mathbf{V}_n\boldsymbol{\alpha}_{nn}.
\end{aligned} \tag{16.28}$$

This system of equations can now be solved for $\mathbf{V}_1, \mathbf{V}_2, \ldots, \mathbf{V}_n$ by repeated substitution and successive applications of Eq. (16.17) in the following manner. Whenever an equation is of the form $\mathbf{V}_i = \mathbf{V}_j\boldsymbol{\alpha}_{ji} + \mathbf{V}_k\boldsymbol{\alpha}_{ki}$ or $\mathbf{V}_i = \mathbf{V}_j\boldsymbol{\alpha}_{ji} + \mathbf{V}_k\boldsymbol{\alpha}_{ki} + \boldsymbol{\lambda}$, where $i \neq j \neq k$, then $\mathbf{V}_i$ can be substituted into all other equations to yield a system with fewer equations and unknowns. Whenever an equation has the form $\mathbf{V}_i = \mathbf{V}_i\boldsymbol{\alpha}_{ii} + \mathbf{V}_j\boldsymbol{\alpha}_{ji}$ (plus $\boldsymbol{\lambda}$ if appropriate), then Eq. (16.17) can be applied to yield $\mathbf{V}_i = \mathbf{V}_j\boldsymbol{\alpha}_{ji}(\boldsymbol{\alpha}_{ii})^*$, which can now be substituted for $\mathbf{V}_i$ in the other equations. Note that, since the graph is assumed to contain no λ-transitions, the condition in Theorem 16.3 that $\boldsymbol{\alpha}_{ii}$ should not contain $\boldsymbol{\lambda}$ can always be met. This procedure will finally lead to a single equation in one variable. This variable can in turn be determined by another application of Eq. (16.17).

The set of strings recognized by a given graph can be described by the union of the **V**'s that correspond to accepting vertices. For example, if vertices B and C in the graph of Fig. 16.18 were accepting vertices then the set of strings recognized by the graph could be described by $\mathbf{B} + \mathbf{C} = (\mathbf{0} + \mathbf{0}(\mathbf{1} + \mathbf{00})^*\mathbf{1})^*\mathbf{0}(\mathbf{1} + \mathbf{00})^*(\boldsymbol{\lambda} + \mathbf{0})$.

Clearly, any system of equations of the form Eq. (16.28) can be uniquely solved by the procedure just outlined, provided that we prove that each of the $\mathbf{V}_i$'s and $\boldsymbol{\alpha}_{ij}$'s is a regular expression. This proof is given in the following theorem.

Theorem 16.4 *The set of strings that take a finite-state machine M from an arbitrary state S_i to another state S_j is a regular set.*

Proof Let Q be any subset of the states of M containing both S_i and S_j, and let R_{ij}^Q denote the set of strings that take the machine from state S_i to state S_j without passing through any state that is outside Q. Since Q may consist of all the states in M, the theorem will be proved if we can show that R_{ij}^Q is regular. The proof will be by induction on the number of states in Q.

Basis Suppose that Q consists of just a single state, which we shall call S_i. Then the set of strings that take S_i into itself without passing through any other state consists of only a finite number of single input symbols. Since by definition each such input symbol is regular, the above set of strings is regular. The corresponding regular expression will be denoted $\mathbf{T}_{ii}$.

Induction step Assume that R_{ij}^Q is regular for all subsets of states containing m or fewer states. Thus, R_{ij}^Q can be described by the regular expression $\mathbf{R}_{ij}^Q$. We shall now prove that the set of strings R_{ij}^P is also regular, where P is a set containing $m+1$ states, including the states S_i and S_j. Suppose now that we remove state S_i from P. The resulting subset consists of only m states and will be referred to as Q; the theorem is assumed to hold for this subset.

Consider a string from R_{ij}^P. In general, it will cause the machine to go through state transitions as follows:

$$S_i, S_t, \ldots, S_u, S_i, \ldots, S_i, \ldots, S_j$$

where the ellipses correspond to transitions within set Q and therefore do not contain occurrences of S_i. The substrings that take the machine from S_i and back into S_i may consist of either single input symbols from the regular set $\mathbf{T}_{ii}$ or of sequences of symbols that take the machine from S_i through some states, say $S_t, \ldots, S_u$, and back into S_i. Such an input sequence actually consists of a single symbol, denoted T_{it}, that takes M from S_i to S_t followed by a sequence from $\mathbf{R}_{tu}^Q$ and ending with a symbol T_{ui} that returns M to S_i. Each of the symbols T_{it} and T_{ui} is clearly regular and, consequently, the set of strings that take M from S_i into S_i can be described by the regular expression

$$\mathbf{T}_{ii} + \sum_{tu} \mathbf{T}_{it} \mathbf{R}_{tu}^Q \mathbf{T}_{ui},$$

where the sum is taken over all possible pairs of states in Q. In addition, since the machine can be taken an arbitrary number of times from S_i through states in Q and back into S_i, the set of corresponding strings can be described by the regular expression

$$\left(\mathbf{T}_{ii} + \sum_{tu} \mathbf{T}_{it} \mathbf{R}_{tu}^Q \mathbf{T}_{ui} \right)^*$$

This set of strings is followed by the set of substrings that take the machine from S_i into S_j. This latter set of substrings consists of all single symbols T_{ij} that take the machine from S_i to S_j and all other strings that take the machine from S_i to S_j via certain states $S_t, \ldots, S_u$. Clearly, this set can be described by the regular expression

$$\mathbf{T}_{ij} + \sum_{tu} \mathbf{T}_{it} \mathbf{R}_{tu}^{Q} \mathbf{T}_{uj}.$$

Consequently, the set of strings R_{ij}^{P} is regular and can be described by the expression

$$\mathbf{R}_{ij}^{P} = \left(\mathbf{T}_{ii} + \sum_{tu} \mathbf{T}_{it} \mathbf{R}_{tu}^{Q} \mathbf{T}_{ui} \right)^{*} \left(\mathbf{T}_{ij} + \sum_{tu} \mathbf{T}_{it} \mathbf{R}_{tu}^{Q} \mathbf{T}_{uj} \right).$$

◇

Combining Theorems 16.2 and 16.4, we obtain the following general result, which is known as Kleene's theorem.

- A finite-state machine recognizes a set of strings if and only if it is a regular set.

Applications

The correspondence between regular sets and finite-state machines enables us to determine whether certain sets are regular. For example, let $\mathbf{R}$ denote a regular set on an alphabet A that can be recognized by a (Moore) machine M_1. Define the complement of $\mathbf{R}$, denoted $\mathbf{R}'$, as the set containing all the strings on A that are not contained in $\mathbf{R}$. The set $\mathbf{R}'$ is regular, since it can be recognized by a machine M_2 that is obtained from M_1 by complementing the output values associated with the states of M_1.

As another example, let us define the intersection of two sets, $\mathbf{P}$ and $\mathbf{Q}$, denoted $\mathbf{P}\&\mathbf{Q}$, as the set consisting of all the strings that are contained in both $\mathbf{P}$ and $\mathbf{Q}$. We can show that the set $\mathbf{P}\&\mathbf{Q}$ is regular by observing that each of the sets $\mathbf{P}'$ and $\mathbf{Q}'$ is regular and, consequently, $\mathbf{P}' + \mathbf{Q}'$ and $(\mathbf{P}' + \mathbf{Q}')'$ are regular. In addition, since $\mathbf{P}\&\mathbf{Q} = (\mathbf{P}' + \mathbf{Q}')'$, the set $\mathbf{P}\&\mathbf{Q}$ is regular. Regular expressions containing the complementation and intersection operations as well as union, concatenation, and closure are called *extended regular expressions*.

The added operations increase our versatility in describing regular sets. For example, consider the set of strings on the alphabet $\{0, 1\}$ such that no string in the set contains three consecutive 0's. This set can be described by the expression $[(\mathbf{0} + \mathbf{1})^{*}\mathbf{000}(\mathbf{0} + \mathbf{1})^{*}]'$, whereas a more complicated expression, such as $(\mathbf{1} + \mathbf{01} + \mathbf{001})^{*}(\boldsymbol{\lambda} + \mathbf{0} + \mathbf{00})$, would be required if the complementation operation were not used. However, since expressions containing the complementation and intersection operations are difficult to manipulate or transform to the corresponding graphs, their usefulness is limited.

The following example will illustrate some additional techniques that can be used to determine whether certain sets are regular.

Example Let M be a finite-state machine whose input and output alphabets are $\{0, 1\}$. Assume that the machine has a designated starting state. Let $z_1 z_2 \cdots z_n$ denote the output sequence produced by M in response to the input sequence $x_1 x_2 \cdots x_n$. Define a set S_M that consists of all the strings w such that $w = z_1 x_1 z_2 x_2 \cdots z_n x_n$, for any $x_1 x_2 \cdots x_n$ in $(\mathbf{0} + \mathbf{1})^*$. Prove that S_M is regular.

Given the state diagram of M, replace each directed arc with two directed arcs and a new state, as shown in Fig. 16.19. Retain the original starting state and designate all the original states as accepting states. The resulting nondeterministic transition graph recognizes the set S_M. Therefore, S_M must be regular.

Fig. 16.19 Illustration of the procedure for designing a recognizer for S_M.

This procedure will now be applied to find a deterministic machine that recognizes the set S_N, where N is the machine described in Fig. 16.20. Replacing every arc of the machine N with two directed arcs, and following the procedure just outlined, we arrive at the transition graph in Fig. 16.21*a*. Converting this graph into deterministic form yields the state diagram of Fig. 16.21*b*.

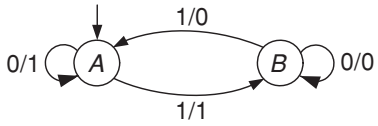
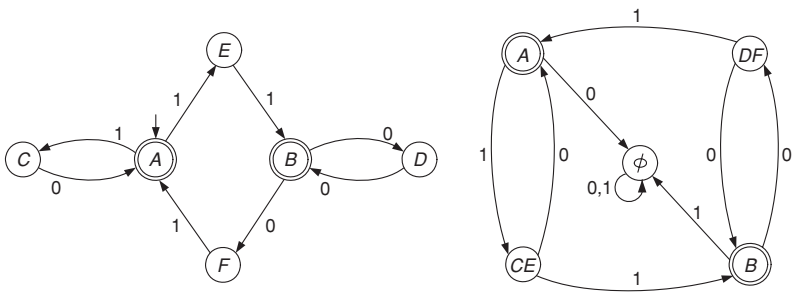


Fig. 16.20 Machine N.



(*a*) Transition graph. (*b*) Equivalent deterministic form.

Fig. 16.21 Constructing a finite-state machine that recognizes S_N.

*16.7 Two-way recognizers

In Section 16.1, we introduced the concept of a recognizer as a finite-state control coupled through a head to a linear input tape. We assumed that the recognizer could move its head in only one direction, to the right. In an attempt to generalize the model further, we will consider recognizers that are not confined to a strict forward motion but can move two ways on their input tapes, that is, to the right and left. A natural question that now arises is whether the option given to the machine to move left and reexamine the input tape increases its computational capabilities. In other words, what characterizes the sets of tapes that are recognized by this class of machines? As we shall see, machines that can move both ways but *cannot* change the tape symbols are no more (nor less) powerful than machines that can move in only one direction.

Description of the model

A *two-way recognizer*, or *two-way machine*, consists of a finite-state control coupled through a head to a tape. Initially, the finite-state control is in its designated starting state, with its head scanning the leftmost square of the tape. The machine then proceeds to read the symbols of the tape one at a time. In each cycle of computation, the machine examines the symbol currently scanned by the head, shifts the head one square to the right or left, and then enters a new (not necessarily distinct) state.

If, when operating in this manner on a given tape, the machine eventually *moves off* the tape at the right-hand end and at that time enters an accepting state, then we shall say that the tape is *accepted* by the machine. A machine can *reject* a tape either by moving off its right-hand end while entering a rejecting state or by looping within the tape. As in the case of one-way machines, the set of tapes that are accepted by a given two-way machine is said to be *recognized* by that machine. The null string λ can be represented either by the absence of an input tape or by a completely blank tape. A machine accepts λ if and only if its starting state is an accepting state.

It is convenient to supply the two-way machine with a new symbol, ¢, called a *left-end marker*, which is entered in the leftmost square of the tape and prevents the head from moving off the left-hand end of the tape. The end marker is not a symbol of the machine's alphabet and must not appear on any other square within the tape.

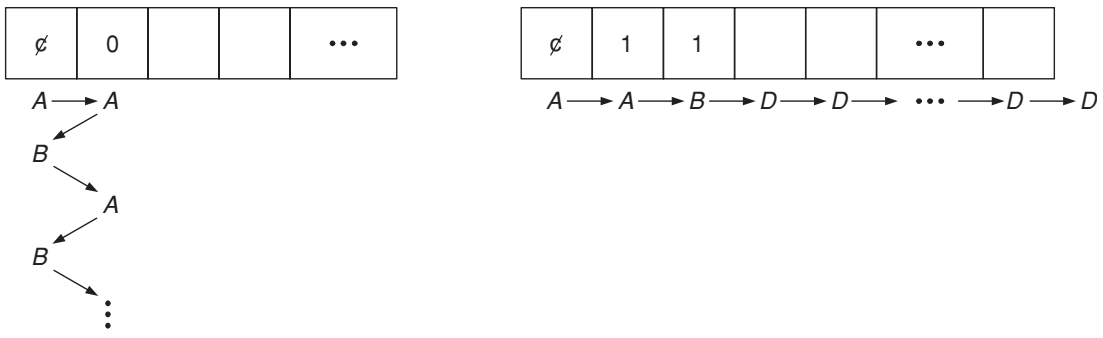
A two-way machine can be described by a state table (or diagram) that specifies, for every possible combination of present state and tape symbol being scanned, the next state that the machine should assume and the direction in which the head is to move. As directional entries, we use the letters L to denote a shift to the left and R to denote a shift to the right.

Example Table 16.1 describes a two-way machine having four states and two tape symbols, 0 and 1, plus the ¢ marker. The starting state is A and the accepting state is C. A blank tape entry indicates that the corresponding state-symbol combination cannot occur. Figure 16.22*a* illustrates the computation that the machine will perform when supplied with a tape that starts with the symbols ¢0. The computation begins with the machine in state A and with its head scanning the left-end marker. According to the state table, the machine will move one square to the right while remaining in state A. The machine will then be scanning a 0 and, consequently, will enter state B and move one square to the left. From now on, the machine will oscillate between these two squares and thus all strings beginning with a 0 will be rejected.

Table 16.1 A two-way-machine recognizing set $\mathbf{100}^*$

	¢	0	1
A	A, R	B, L	B, R
B	A, R	C, R	D, R
C		C, R	D, R
D		D, R	D, R

Next, suppose that the machine is presented with a tape that starts with ¢11. The computation is illustrated in Fig. 16.22*b*. When the third symbol is reached, the machine is in state D. Thereafter, it remains in state D regardless of the tape content until it moves off the tape. Since D is a rejecting state, all sets of tapes starting with 11 are rejected. Finally, let the tape consist of the string ¢10. Again, the machine starts by moving to the right, and it goes through a succession of states until it moves off the tape in state C. Since C is an accepting state, the tape in question is accepted. By similar reasoning, we can verify that the machine recognizes the set $\mathbf{100}^*$.



(*a*) A loop. (*b*) Rejection of a tape.

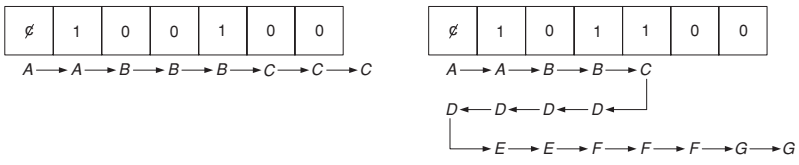
Fig. 16.22 Illustration of computations.

In the next section we shall prove that two-way machines are as powerful as one-way machines with respect to the classes of tapes that they can recognize. For some computations, however, it is convenient to use two-way recognizers since they may require fewer states than the equivalent one-way recognizers. However, for the ability of a two-way machine to reverse direction and reread its tape, we pay in terms of an increased computation time.

Example Consider the two-way machine shown in Table 16.2, which accepts a tape if and only if it contains at least three 1's and at least two 0's. The starting and accepting states are A and G, respectively. Some typical computations are shown in Fig. 16.23. The operation of the machine can be summarized as follows. Initially the machine is in state A and the head is scanning the left-end marker. The head then proceeds to the right to determine whether the tape contains at least three 1's. If the tape contains two or fewer 1's, it is rejected; if it contains three 1's then the head reverses its direction and moves left until it again reaches the left-end marker. The machine then proceeds to the right to determine whether the tape contains two or more 0's. If it does, the machine enters state G and will eventually accept the tape; otherwise the tape will be rejected.

Table 16.2 A two-way machine

	¢	0	1
A	A, R	A, R	B, R
B		B, R	C, R
C		C, R	D, L
D	E, R	D, L	D, L
E		F, R	E, R
F		G, R	F, R
G		G, R	G, R



(*a*) Rejecting a tape. (b) Accepting a tape.

Fig. 16.23 Example of computations.

The minimal one-way machine that is equivalent to the two-way machine in Table 16.2 has 12 states. This larger number of states is necessary because of the way in which a one-way machine operates. Any one-way machine that recognizes the above set of tapes must examine the tapes for the proper

number of 0's and 1's *simultaneously*. This can be done, for example, by the use of two separate counters, one for the 1's and the other for the 0's. The state of the machine in such a case is the composite state of the two counters. Consequently, the number of states required to perform the above computation is proportional to the *product* of the numbers of states required to test the tapes for the number of 0's and the number of 1's separately. The two-way machine in this example tests the tapes first for the appropriate number of 1's and then for the appropriate number of 0's. Thus, the number of states is proportional to the *sum* of the numbers of states required to test the tapes for the two requirements separately.

Conversion to one-way recognizers

We now turn to proving that two-way machines can recognize sets of tapes (or strings) if and only if they are regular sets. Specifically, we shall show that for every given two-way machine there is an equivalent one-way machine that recognizes the same set of tapes. Since the details of the construction procedure do not add significantly to its understanding, we shall confine our discussion to sketching the main ideas of the proof.

Since a one-way machine makes as many moves as there are symbols on the tape while a two-way machine can make moves by reversing direction, the one-way machine cannot keep track of all the moves of the two-way machine or simulate them. It is, therefore, necessary to isolate the significant information gained by a two-way machine on moving to the left from the particular sequence of moves. Consider an initial segment at the left of the input tape, and suppose that the head is scanning the rightmost square of this segment. The only way in which this segment can influence the future behavior of the two-way machine is via the state which the machine is in when (and if) it leaves this segment. Thus, when a two-way machine backs up and reexamines a segment of the tape, the state S_i in which the machine reenters the segment and the corresponding state S_i' which the machine would be in if it left the segment are the only two factors of significance in predicting the future behavior of the machine.

A two-way machine having n states can be in any of these states when it scans the rightmost square of the initial segment. Two cases must be considered. First, the machine may never leave the segment but oscillate within it. Second, the machine will ultimately leave the segment on the right in one of its n states. Thus, a reentry into a segment may have $n + 1$ outcomes, that is, leaving the segment in one of the n states or not leaving it. Consequently the effect of the segment on the computation can be determined by specifying, for each state S_i in which the machine might reenter the segment, which of the $n + 1$ outcomes would indeed result. Such a specification is accomplished by means of a *crossing function* (or *crossing table*), denoted $C(S)$.

Table 16.3 A two-way machine M

	¢	0	1
A	A, R	B, R	C, R
B		A, R	A, L
C		B, R	D, L
D		C, L	B, R

Table 16.4 Crossing functions for M

S_i	$C(S_i)$ for ¢001	$C(S_i)$ for ¢0011
A	C	C
B	0	0
C	C	0
D	B	B

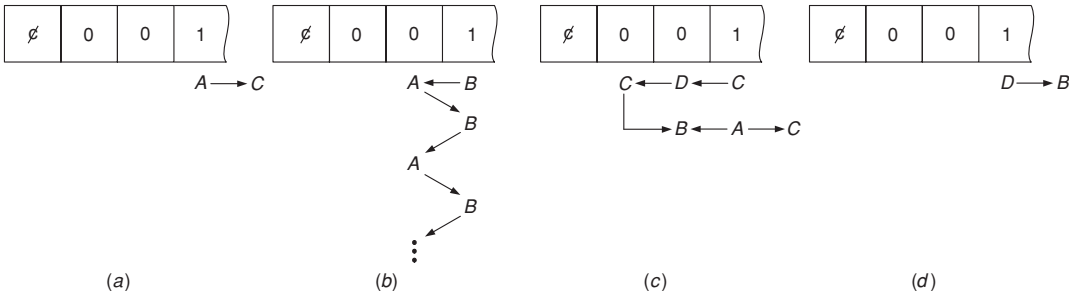


Fig. 16.24 Computations on the segment ¢001.

The following is extracted from Shepherdson's proof [11]. It summarizes the informal arguments in support of his proof. (Note that M denotes the given two-way machine and t denotes an initial tape segment.)

If we think of the different states which M could be in when it reentered t as the different questions M could ask about t, and the corresponding states M would be in when it subsequently left t again, as the answers, then we can state the result more crudely and succinctly thus: A machine can spare itself the necessity of coming back to refer to a piece of tape t again, if, before it leaves t, it thinks of all the possible questions it might later come back and ask about t, answers these questions now and carries the table of question–answer combinations forward along the tape with it, altering the answers where necessary as it goes along.

As an example, consider the two-way machine M given in Table 16.3 and the initial tape segment ¢001. The starting and accepting states are A and C, respectively. Figure 16.24 illustrates, for each possible initial state, the computation performed by the machine if its head is initially scanning the rightmost symbol of the given segment. If the initial state is A than the machine immediately leaves the segment in state C. If, however, the initial state is B then the machine will oscillate between states B and A and will never leave the segment. From Fig. 16.24 we can derive the crossing function associated with the segment ¢001, as shown in the first two columns of Table 16.4. The first column, S_i, of this table lists the states of the machine while the second column, $C(S_i)$, lists the states in which the machine crosses the given segment to the right. An entry 0 indicates that the tape will be rejected.

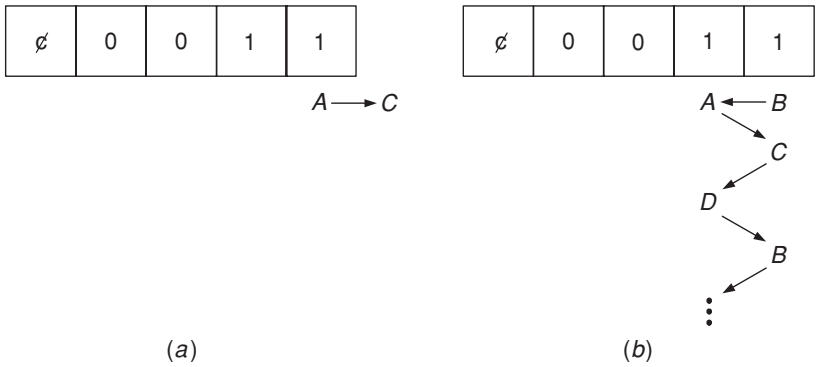


Fig. 16.25 Illustration of computations on the segment ¢0011.

An important property of crossing functions is that the crossing function of a $(k+1)$-symbol segment can be obtained from the crossing function of a k-symbol segment. The rightmost column of Table 16.4 contains the crossing function associated with the segment ¢0011. This crossing function can be obtained from the crossing function of the segment ¢001. Suppose, for example, that the machine is in state A and is scanning the rightmost symbol of ¢0011. According to the state table in Table 16.3, the machine will move to the right and enter state C, as illustrated in Fig. 16.25*a*. Accordingly, the entry in row A in the rightmost column is C. If, however, the machine is in state B while scanning the rightmost symbol of the given segment then it will move left and enter state A. According to the crossing function associated with the segment ¢001, the machine will leave this segment in state C, as shown in Fig. 16.25*b*. Again it will scan the rightmost symbol of ¢0011 and, according to the state table, again it will move left and enter state D. According to the crossing function for ¢001, the machine will ultimately leave this segment on the right and enter state B. Evidently such a sequence of moves indicates that the computation will never halt and, consequently, a 0 is entered in row B of Table 16.4. The same line of reasoning leads to the specification of the entries in rows C and D.

The procedure followed in this example leads to the conclusion that, *given the crossing functions associated with the initial segments containing k symbols, we can readily obtain the crossing functions associated with all initial segments containing $k+1$ symbols*. In fact, since the number of distinct crossing functions associated with a specific two-way machine cannot exceed $(n+1)^n$, where n is the number of states, it is possible to construct a one-way machine that will read the tape from left to right and compute with each move the crossing function associated with the corresponding initial segment. Such a machine will have as many states as there are crossing functions. Its input alphabet is the same as that of the corresponding two-way machine. The next-state entries of the one-way machine are obtained as follows. For a given state, which corresponds to a crossing function of the two-way machine, the next-state entry under the input symbol α corresponds to the new crossing function obtained from the given one and the symbol α, as illustrated in Fig. 16.25.

Once we have a one-way machine that scans the tape from left to right and computes the crossing functions associated with successive initial segments, since the starting state of the two-way machine is specified it is a simple matter to determine, after each move of the one-way machine, the corresponding next state of the two-way machine. Consequently, we can determine the state of the two-way machine when it moves off the tape. If this state is an accepting state then the one-way machine will also accept the tape; otherwise it will reject the tape. We thus have the following result.

- The sets of strings recognized by two-way finite-state machines are the same as the sets recognized by one-way finite-state machines. Moreover, there exists an effective procedure for constructing a one-way machine that recognizes the same set of strings as a given two-way machine.

Although two-way machines are no more powerful than one-way machines with respect to the sets of strings that they can recognize, it is often more convenient to describe certain computations in terms of two-way machines. The equivalence of the two models, however, makes it generally possible to use either.

Notes and references

Nondeterministic graphs were first used by Myhill [8] and further developed by numerous investigators, in particular those working on languages. The initial concept of regular expressions and the equivalence between regular expressions and finite-state machines were presented by Kleene [5]. Simpler techniques for converting regular expressions into transition graphs, and vice versa, were subsequently developed by Copi, Elgot, and Wright [4], McNaughton and Yamada [6], and Ott and Feinstein [9]. The procedure presented in this chapter of constructing transition graphs from regular expressions is due to Ott and Feinstein [9], while the procedure used to derive regular expressions that describe transition graphs is due to Arden [1]. A survey of regular expressions is available in Brzozowski [2].

Two-way machines were first investigated by Rabin and Scott [10], who provided the first proof that two-way machines are equivalent to one-way machines. Shepherdson [11] subsequently provided a simpler proof, the one outlined in Section 16.7.

[1] Arden, D. N.: "Delay logic and finite state machines," in *Proc. Second Ann. Symp. Switching Theory and Logical Design*, pp. 133–151, October 1961.

[2] Brzozowski, J. A.: "A survey of regular expressions and their applications," *IRE Trans. Electron. Computers*, vol. EC-11, pp. 324–335, June 1962.

[3] Brzozowski, J. A.: "Derivatives of regular expressions," *J. Assoc. Computing Machinery*, vol. 11, pp. 481–494, 1964.

[4] Copi, I. M., C. C. Elgot, and J. B. Wright: "Realization of events by logical nets," *J. Assoc. Computing Machinery*, vol. 5, pp. 181–196, April 1958; reprinted in Moore [7].

[5] Kleene, S. C.: *Representation of Events in Nerve Nets and Finite Automata*, pp. 3–41, Automata Studies, Princeton University Press, 1956.
[6] McNaughton, R., and H. Yamada: "Regular expressions and state graphs for automata," *IRE Trans. Electron. Computers*, vol. EC-9, pp. 39–47, March 1960; reprinted in Moore [7].
[7] Moore, E. F. (ed.): *Sequential Machines: Selected Papers*, Addison-Wesley, Reading MA, 1964.
[8] Myhill, J.: "Finite automata and the representation of events," WADC Technical Report 57–624, pp. 112–137, 1957.
[9] Ott, G. H., and N. H. Feinstein: "Design of sequential machines from their regular expressions," *J. Assoc. Computing Machinery*, vol. 8, pp. 585–600, October 1961.
[10] Rabin, M. O., and D. Scott: "Finite automata and their decision problems," *IBM J. Res. Develop.*, vol. 3, no. 2, pp. 114–125, April 1959; reprinted in Moore [7].
[11] Shepherdson, J. C.: "The reduction of two-way automata to one-way automata," *IBM J. Res. Develop.*, vol. 3, no. 2, pp. 198–200, April 1959; reprinted in Moore [7].

Problems

Problem 16.1. For each of the sets described as follows, find a transition graph that recognizes the set.
(a) The set of strings on the alphabet $\{0, 1\}$ that start with 01 and end with 10.
(b) The set of strings on the alphabet $\{0, 1\}$ that start and end with a 1, and in which every 0 is immediately preceded by at least two 1's.
(c) The set of strings on the alphabet $\{0, 1, 2\}$ in which every 2 is immediately followed by exactly two 0's and every 1 is immediately followed by either 0 or else by 20.

Problem 16.2. Consider the class of transition graphs containing no λ-transitions.
(a) Show a procedure for converting a specified transition graph with several starting vertices into a graph with just one starting vertex. Apply your procedure to the graph in Fig. P16.2.
Hint: Add a new vertex and designate it as the starting vertex.
(b) Show a procedure for converting a given transition graph with several accepting vertices into a graph with just one accepting vertex. Apply your procedure to the graph in Fig. P16.2.
(c) Is it always possible to convert an arbitrary transition graph into a graph with just one starting vertex and just one accepting vertex? Determine the conditions under which such a conversion is possible.

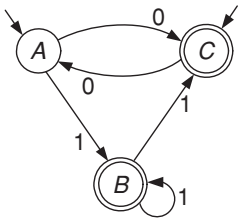
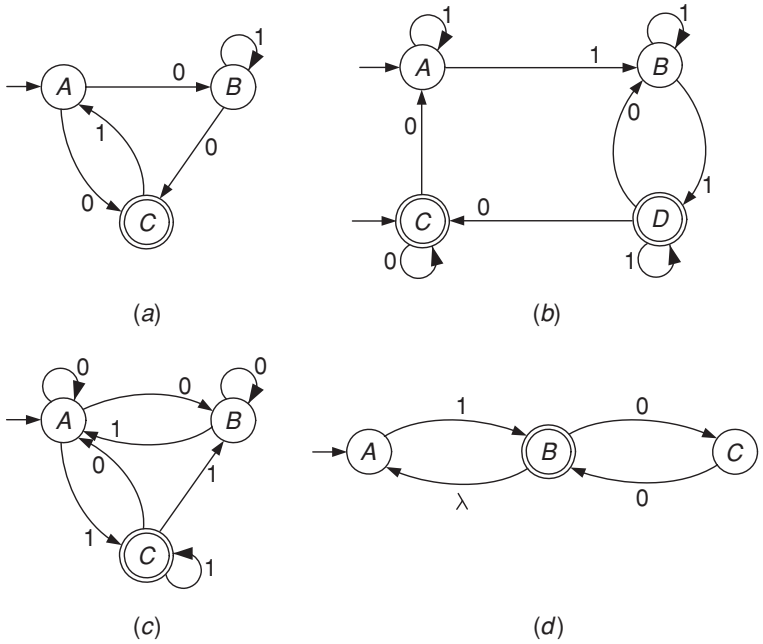


Fig. P16.2

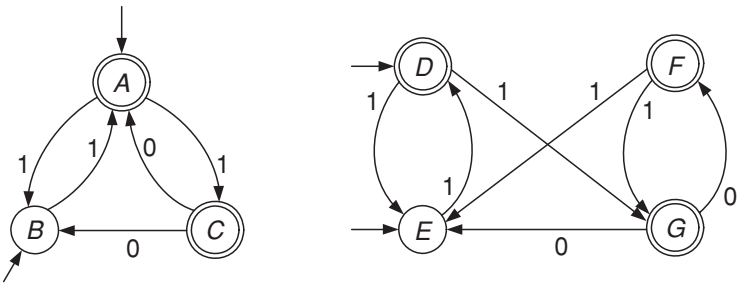
Problem 16.3. For each of the nondeterministic graphs in Fig. P16.3, find an equivalent deterministic graph (in standard form) that recognizes the same set of strings.

Fig. P16.3



Problem 16.4. Show that the two graphs in Fig. P16.4 are equivalent by converting them to deterministic forms.

Fig. P16.4



Problem 16.5. Design a finite-state machine that accepts only those input sequences that end with either 101 or 0110. First construct a nondeterministic graph that recognizes the above set of sequences and then convert this graph into an equivalent deterministic graph. Discuss the merits of this approach versus the direct approach of deriving a state diagram from a word description.

Problem 16.6. Give a word description of the sets described by the following regular expressions:

(a) $\mathbf{110}^*(\mathbf{0}+\mathbf{1})$;

(b) $\mathbf{1}(\mathbf{0}+\mathbf{1})^*\mathbf{101}$;

(c) $(\mathbf{10})^*(\mathbf{01})^*(\mathbf{00}+\mathbf{11})^*$;

(d) $(\mathbf{00}+(\mathbf{11})^*\mathbf{0})^*\mathbf{10}$.

Problem 16.7. Find a regular expression for each set described in Problem 16.1.

Problem 16.8. Use the identities in Section 16.4 to verify the identities below:
(a) $\mathbf{10} + (\mathbf{1010})^*[\boldsymbol{\lambda}^* + \boldsymbol{\lambda}(\mathbf{1010})^*] = \mathbf{10} + (\mathbf{1010})^*$;
(b) $(\mathbf{0}^*\mathbf{01} + \mathbf{10})^*\mathbf{0}^* = (\mathbf{0} + \mathbf{01} + \mathbf{10})^*$;
(c) $\boldsymbol{\lambda} + \mathbf{0}(\mathbf{0} + \mathbf{1})^* + (\mathbf{0} + \mathbf{1})^*\mathbf{00}(\mathbf{0} + \mathbf{1})^* = [(\mathbf{1}^*\mathbf{0})^*\mathbf{01}^*]^*$.

Problem 16.9.
(a) Use the induction procedure developed in Section 16.5 to find a transition graph that recognizes the set of strings described by

$$\mathbf{R} = \mathbf{0}(\mathbf{11} + \mathbf{0}(\mathbf{00} + \mathbf{1})^*)^*.$$

(b) Convert the graph found in (a) to a deterministic state diagram.

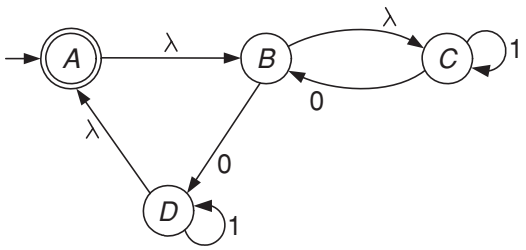
Problem 16.10. For each of the following expressions, find a transition graph that recognizes the corresponding set of strings:
(a) $(\mathbf{0} + \mathbf{1})(\mathbf{11} + \mathbf{0}^*)^*(\mathbf{0} + \mathbf{1})$;
(b) $(\mathbf{1010}^* + \mathbf{1}(\mathbf{101})^*\mathbf{0})^*\mathbf{1}$;
(c) $(\mathbf{0} + \mathbf{11})^*(\mathbf{1} + (\mathbf{00})^*)^*\mathbf{11}$.

Problem 16.11. The regular expression that corresponds to the transition graph in Fig. P16.11 is

$$\mathbf{R} = [(\mathbf{1}^*\mathbf{0})^*\mathbf{01}^*]^*.$$

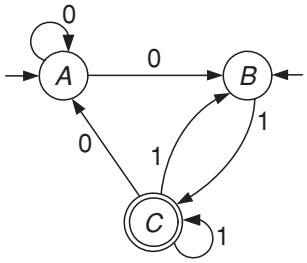
Find a finite-state machine that recognizes the same set of strings.

Fig. P16.11



Problem 16.12. The nondeterministic graph in Fig. P16.12 has A and B as starting vertices and C as an accepting vertex.
(a) Find a regular expression that describes the set of strings accepted by this graph.
(b) Derive a reduced deterministic machine equivalent to this graph.

Fig. P16.12



Problem 16.13. For each machine in Table P16.13, find a regular expression that describes the set of input strings recognized by the machine. In each case the starting state is A.

Table P16.13

	NS		
PS	$x=0$	$x=1$	z
A	A	B	0
B	B	A	1

(a)

	NS		
PS	$x=0$	$x=1$	z
A	B	A	1
B	B	C	0
C	A	B	1

(b)

	NS, z	
PS	$x=0$	$x=1$
A	$B, 0$	$A, 1$
B	$A, 1$	$C, 1$
C	$C, 0$	$B, 0$

(c)

Problem 16.14. Find a regular expression on the alphabet $\{0, 1, 2\}$ for the set of strings recognized by the graph of Fig. P16.14.

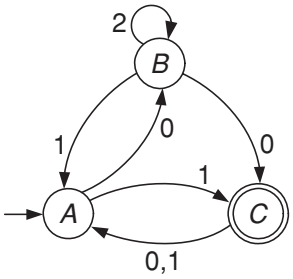


Fig. P16.14

Problem 16.15. Determine whether each of the following sets on the alphabet $\{0, 1\}$ is regular and justify your answer:

(a) the set consisting of those strings that contain, for all k, k 1's and $k+1$ 0's;
(b) the set of strings in which every 0 is immediately preceded by *at least* k 1's and is immediately followed by *exactly* k 1's, where k is a specified integer;
(c) the set of strings that contain more 1's than 0's;
(d) the set of strings consisting of a block of k^2 0's immediately followed by a single 1, where $k = 0, 1, 2, \ldots$

Problem 16.16

(a) Let M be a deterministic Mealy-type finite-state machine with a starting state A. Prove that if T is the set of strings that can be produced as output strings by M then T is a regular set. Find a procedure to design a finite-state machine that will recognize T.
Hint: Use the output successor table of M.
(b) Apply your procedure to find a finite-state machine that will recognize the set of output strings that can be produced by the machine defined by Table P16.16.

Table P16.16

	NS, z	
PS	$x = 0$	$x = 1$
A	$B, 1$	$A, 1$
B	$A, 0$	$C, 0$
C	$D, 1$	$B, 0$
D	$C, 0$	$A, 1$

Problem 16.17. The reverse $\mathbf{R}^{\mathrm{r}}$ of a set $\mathbf{R}$ is the set that consists of the reverses of the strings in $\mathbf{R}$. Thus, for example, if 0101 is in $\mathbf{R}$ then 1010 is in $\mathbf{R}^{\mathrm{r}}$.

(a) Prove that if $\mathbf{R}$ is regular then so is $\mathbf{R}^{\mathrm{r}}$.
Hint: Develop a systematic procedure to convert a given regular expression into its reverse.

(b) Apply the above procedure to find the reverse of the expression

$$\mathbf{R} = (\mathbf{00})^*(\mathbf{0} + \mathbf{10}^*)^* + \mathbf{10}^*(\mathbf{01}^*\mathbf{10}^*)^*.$$

Problem 16.18. Either prove each of the following statements or show a counter example.

(a) Every *finite* subset of a nonregular set is regular.

(b) The expressions $\mathbf{P} = (\mathbf{1}^*\mathbf{0} + \mathbf{001})^*\mathbf{01}$ and $\mathbf{Q} = (\mathbf{1}^*\mathbf{001} + \mathbf{00101})^*$ are equivalent.

(c) Let $\mathbf{R}$ denote a regular set. Then the set consisting of all the strings in $\mathbf{R}$ that are identical to their own reverses is also a regular set.

(d) Every subset of a regular set is also regular.

Problem 16.19. Consider the nondeterministic machine M^{n}, which is obtained from a strongly connected deterministic machine M by interchange of the sets of starting and accepting states and reversal of the arrows on the state diagram.

(a) If the machine M recognizes the set $\mathbf{R}$, what is the set recognized by M^{n}?

(b) Prove that the deterministic machine obtained by applying "subset construction" to M^{n} has no equivalent states.

Problem 16.20. Let $\mathbf{P}$ be a regular set consisting of strings of even length. Define a set $\mathbf{Q}$ that consists of exactly those strings that can be formed by taking the first half of each member of $\mathbf{P}$. (For example, if 10110100 is contained in $\mathbf{P}$ then 1011 will be contained in $\mathbf{Q}$.) Prove that $\mathbf{Q}$ is a regular set.

Hint: Design a machine that recognizes $\mathbf{Q}$.

Problem 16.21. Let $\mathbf{P}$ be a regular set, and let $\mathbf{Q}$ be the set formed of all the strings from $\mathbf{P}$ with even-numbered symbols deleted; that is, if $a_1a_2a_3a_4a_5\cdots$ is a string in $\mathbf{P}$, then $a_1a_3a_5\cdots$ is a string in $\mathbf{Q}$. Prove that $\mathbf{Q}$ is a regular set.

Problem 16.22. Let $\mathbf{P}$ be an arbitrary regular set. Consider those strings w in $\mathbf{P}$ such that both w and ww are in $\mathbf{P}$. Define $\mathbf{Q}$ to be the set consisting of all the above w's. Thus, for example, if 101 and 101101 are in $\mathbf{P}$ then 101 is in $\mathbf{Q}$. Prove that $\mathbf{Q}$ is a regular set.

Problem 16.23. Let $\mathbf{R}$ be a regular set on the alphabet $\{0, 1\}$. The *derivative of* $\mathbf{R}$ *with respect to* x, denoted $\mathbf{R}_x$, is defined as the set consisting of all substrings y such that xy is in $\mathbf{R}$. For example, if $\mathbf{R} = \mathbf{01}^* + \mathbf{100}^*$ then $\mathbf{R}_0 = \mathbf{1}^*$ and $\mathbf{R}_{10} = \mathbf{0}^*$.

(a) Prove that, for all x, $\mathbf{R}_x$ is a regular set.

(b) Show that there is only a finite number of distinct derivatives for any regular set (although there is an infinite number of choices for x). Find an upper bound on this number if it is known that $\mathbf{R}$ can be recognized by a transition graph with k vertices.

Problem 16.24. The *right quotient* of two sets X and Y, denoted X/Y, is defined as the set Z that consists of all strings z such that $x = zy$ is a string in X and y is a string in Y. Prove that if X is a regular set then $Z = X/Y$ is also a regular set. The set Y may or may not be regular.

Problem 16.25. Determine which of the following tapes is accepted by the two-way machine shown in Table P16.25. The starting and accepting states are A and D, respectively.

(a) ¢010101

(b) ¢010110

(c) ¢10101

Table P16.25

	¢	0	1
A	A, R	B, R	C, R
B		D, L	C, L
C		C, R	D, R
D		B, R	C, L

Problem 16.26. A two-way machine with n states is started at the left end of a tape containing p squares. What is the maximum number of moves that the machine can make before accepting the tape?

Problem 16.27. Construct a two-way machine whose tape may contain symbols from the alphabet {0, 1, 2} plus the left-end marker and which accepts a string if and only if it starts and ends with a 2 and every 2 except the first is immediately preceded by a substring from the set $\mathbf{0}(\mathbf{01})^*$.

Problem 16.28. A given two-way machine recognizes a set of tapes A, rejects a set B, and does not accept (by never halting) a set C. Can a two-way machine be designed so that it:

(a) recognizes B, rejects A, does not accept C?

(b) recognizes A and rejects B and C?

(c) recognizes A but does not accept B and C?

(d) recognizes A and C and rejects B?

(e) recognizes C, rejects B, and does not accept A?

Hint: Determine first which of the sets A, B, and C is regular.