

6 Automatic Test Pattern Generation

6.1 Introduction

In Chapter 1, we explained why fault models are important in making testing more manageable and effective. We also gave a brief description of the three types of testing that are fault-independent: exhaustive, pseudoexhaustive, and pseudorandom. Exhaustive testing is not a realistic approach for large circuits due to its test application time. Pseudoexhaustive testing is more of a design for testability approach than a test pattern generation technique. Pseudorandom test pattern generation is deferred to Chapter 11.

In this chapter we present test pattern generation that targets specific faults. Test sets of this type are called *fault-oriented or deterministic test sets*. The fault model used here is the *stuck-at* type. We concentrate only on the faults detected by observing the logic level (voltage) of the primary outputs at the gate level. Current testing is the topic of Chapter 7.

A deterministic test set may be *algebraic* or *algorithmic*. An example of an algebraic method is the use of the Boolean difference that we introduced in Chapter 3. We applied it to combinational circuits, but, it has been used also for sequential circuits [Hsiao 1971]. However, such a method is not efficient for test pattern generation of large circuits [Larrabee 1989], but it gives a good understanding of path sensitization [Sellers 1968]. It has been shown that test pattern generation for a combinational circuit is an NP-complete problem [Ibarra 1975, Fujiwara 1982]. This suggests that there are no test pattern generation algorithms with polynomial time complexity. Goel argues that the execution time of an algorithm grows proportionately to the square of the number of gates in the circuit [Goel 1980]. Because of such complexity, several heuristics have been proposed. The best known are the D-algorithm [Roth 1967], critical path algorithm, PODEM [Goel 1981], FAN [Fujiwara 1983], and SOCRATES [Schultz 1988]. The D-algorithm is based on D-calculus and guarantees a solution (a test pattern) if one exists. The other algorithms cannot guarantee a solution, but they do work in most situations.

All algorithms are based on the four main operations processes of excitation, sensitization, justification, and implication, all described in the next section. Each algorithm starts from a different part of the circuit. The D-algorithm starts at the faulty line, and its main difficulty is in reconverging fanout through XOR gate as we clarify later in the chapter. The critical path algorithm, which was devised as an alternative to simulation, starts from the

primary outputs of the circuit and generates a test pattern for several faults, while PODEM starts from the primary inputs. FAN is an improvement over PODEM because of the use of new heuristics that minimize on backtracking. Also, SOCRATES is an improvement over FAN and utilizes testability measures to facilitate backtracking.

In the rest of the chapter we discuss the D-algorithm, the critical path, and PODEM. However, we first define path sensitization and justification, which we have used informally in generating patterns in Chapter 2. In describing automatic test pattern generation algorithms, we also make use of graphical representation of the circuit on the gate level.

6.2 Terminology and Notation

In this section we revisit some concepts and define some terms and operations that will be helpful in understanding the deterministic test pattern generation.

6.2.1 Basic Operations

To generate a pattern for a stuck-at fault on a line, we need to *provoke* or *excite* the fault, *sensitize* the results to a primary output, and *justify* the logic values required on the other lines in the circuit. In performing these operations, values are assigned to the lines in the circuits. We need to find the *implications* of these values on other gates.

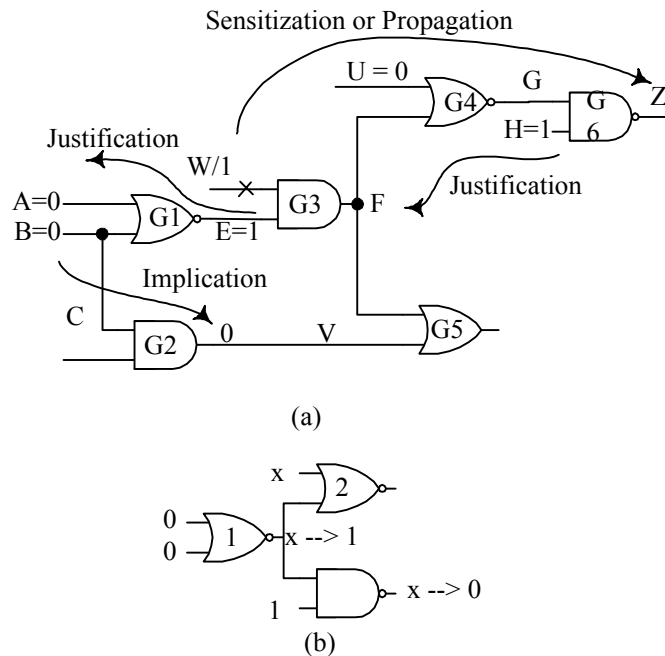


Figure 6.1 Test Pattern Generation Terminology

To *provoke* or *excite* a line is to *control* it to a logic value that is the complement of the value at which it is stuck at; this is equivalent to placing the faulty signal on the line. This signal is a discrepancy from the fault-free circuit. For example, to provoke the stuck-at 1 fault on line W , $W/1$, of the circuit in Fig. 6.1a, we must put 0 on this line, $W = 0$.

It is necessary to *sensitize* or *propagate* the fault to a primary output in order to observe it. The path from the faulty location to the primary output is a *sensitizing* or *propagation path*. A fault may have more than one sensitizing path to the same output or to different outputs. The fault $W/1$ has one sensitizing path: through G3, G4, and G6. To sensitize the fault to the output of G3, we must have $E = 1$. Finally, to propagate the fault to the primary output, Z , we need to have $H = 1$. The values on E and H need to be *justified* to the primary inputs. We justify 1 on E by having $A = B = 0$. Next we find the *implication* of B on gate G2. Sometimes in propagating and justifying we encounter a conflict because some of the lines we need to control have values already assigned. In such cases it is said that we encountered an *inconsistency* [Roth 1966].

6.2.2 Logic and Set Operations

Provoking the fault on a line is equivalent to placing the complement of the faulty signal on the line. Since this faulty signal may be 0 or 1, we distinguish it by denoting the SA0 (SA1) fault by D (D'). This *discrepancy* notation was developed for use in the D-algorithm, but we will also adopt it throughout the chapter independent of the algorithm. The sensitization of the fault from one of the inputs of a gate to its output is determined by the signals on the other inputs of the gate and the type of gate. Before applying any patterns, we assume that all the nodes of the circuit are at an unknown logic value, x . In Fig. 6.1b, the 0 values on the inputs of the first NOR gate change its output from x to 1. The implication of this value on the second NOR is to change its output from x to 0, while the output of the AND gate remains at the unknown value x . Thus the logic values are now extended from $\{0,1\}$ to the set $\{0,1,D,D',x\}$. Table 6.1 defines the three main logic operations on this 5-tuplet of logic values.

During sensitization (propagation toward the output) and justification (tracing backward to the inputs), the same line may be assigned different logic values. Sometimes these values may be compatible; other times, they are not. To decide if we can conciliate the two assignments, we make use of the *intersection* of the two assignment sets that are defined in Table 6.2, where ϕ is the empty set and ψ is an undetermined operation. Consider an assignment that requires, for example, logic 0 on a line and another assignment requiring x on the same line. Since x may be

interpreted as a 0 or a 1, the result of intersecting 0 with x is 0. On the other hand, if the second assignment is 1, there are no values to satisfy both assignments, and we have an inconsistency [Roth 1966].

Table 6.1 Logic operations using the set of Logic Values 0, 1, D , D' , and x .

AND	0	1	x	D	D'
0	0	0	0	0	0
1	0	1	x	D	D'
x	0	x	x	x	x
D	0	D	x	D	0
D'	0	D'	x	0	D'

OR	0	1	x	D	D'
0	0	1	x	D	D'
1	1	1	1	1	1
x	x	1	x	x	x
D	D	1	x	D	1
D'	D'	1	x	1	D'

A	A'
0	1
1	0
x	x
D	D'
D'	D

Table 6.2 The Intersection Operation.

	0	1	x	D	D'
0	0	ϕ	0	Ψ	Ψ
1	ϕ	1	1	Ψ	Ψ
x	0	1	x	D	D'
D	Ψ	Ψ	D	D	ϕ
D'	Ψ	Ψ	D'	ϕ	D'

6.2.3 Fault List

To generate a test set for a circuit, we need first to develop a fault list as defined in Chapter 2. Instead of listing all the possible faults in the circuit, we may use fault equivalence and dominance (see Chapter 2) to collapse the list to the checkpoints of the circuit. For a fanout-free circuit, the checkpoints are the primary inputs. Otherwise, the checkpoints are comprised of the primary inputs and the branches of all fan-outs in the circuit. Circuit S1 in Fig. 6.2 has 12 lines, including the branches of the stem, F. The total number of SSA faults is then 24. There are seven checkpoints: A , B , U , W , Y , $F1$, and $F2$. Therefore, the total number of faults is reduced to 14. However, using equivalence and dominance, we can reduce this list to contain fewer faults. For a large circuit, collapsing faults is not as easily done, but, of course, any reduction in the number of faults is highly desirable.

In the remainder of the chapter we start with the D-algorithm and then present the main features of other heuristics which are an improvement over this algorithm. We assume that a fault list has been compiled for the circuit under test and that this fault list has been collapsed to the shortest possible number.

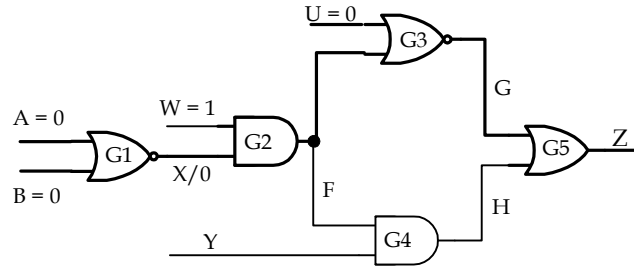


Figure 6.2 A Circuit to Illustrate the Dalgorithm

6.3 The D-Algorithm

The D-algorithm is based on set theory and is explained formally in [Roth 1966]. Instead of using mathematical derivation, we outline it here with the flowchart shown in Fig. 6.3. We illustrate the operations defined in Section 6.2.1 using the D-algorithm terminology. To provoke a fault, we construct its *primitive D-cube for failure* (PDCF). This is simply the set of logic values on the inputs and outputs of this gate that *provoke* the fault on its output. Thus if the gate in question is a three-input AND gate and the fault is an SA0, the PDCF is the set $\{1,1,1,D\}$. This is illustrated in Fig. 6.4. Similarly, for a two-input NOR gate and an SA1 fault, it may be one of the three cubes $\{0,1, D'\}$, $\{1,0, D'\}$, or $\{1,1, D'\}$ since it is sufficient to have a 1 on any of the inputs of the NOR gate to drive the output to 0. The PDCF for $X/0$ is $\{A,B, X\} = \{0,0,D\}$.

The next step is to enumerate the possible sensitizing paths to primary outputs. The PDCF is then propagated along any of these paths, one gate at a time, utilizing the logical operations defined in Table 6.1. Propagating this cube to the output is called the *D-drive*. To propagate the fault on W in Fig. 6.4b, we form the *D-cube* $\{W, E, F\} = \{D, 1, D\}$. For this we use the logical operation in Table 6.1. As we continue propagating the *D-cube*, we will reach a primary output on which a D or D' will appear; then the signals assigned on the nodes during propagation are justified to the primary inputs. One possibility of the justification of F is illustrated in Fig. 6.4c. Both propagation and justification operations require performing the intersection on the cubes according to the operations defined in Table 6.2. Next we illustrate the algorithm by applying it to faults on an internal node, on a primary input, and on a primary output of circuit S1 in Fig. 6.2, which is the same circuit we use to illustrate other ATPG algorithms.

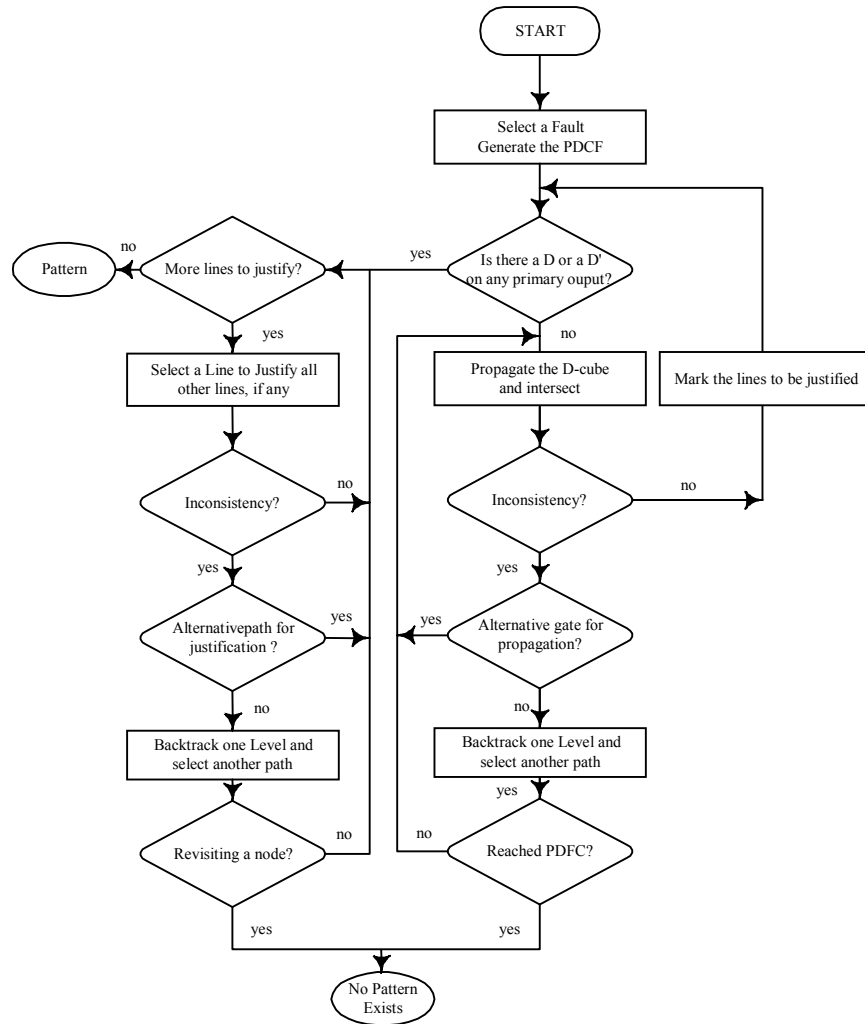


Figure 6.3 D-algorithm

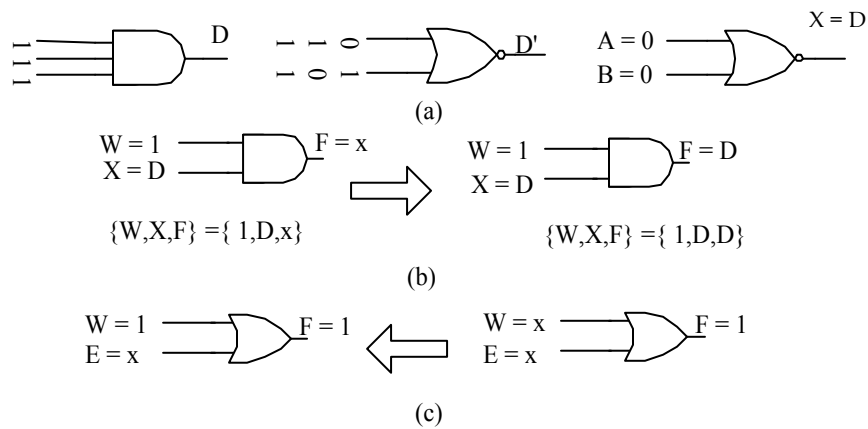


Figure 6.4 D-algorithm Operations a) PDCF, b) Propagating the D-cube, c) Justification

6.3.1 Case of an Internal Node

We selected line X as an illustration for an internal node and we assume that it is stuck at 0. We make a list of all possible paths, which can be sensitized, from the fault to a primary output. There are two sensitization paths: one through $G2$, $G3$, and $G5$ and the other through $G2$, $G4$, and $G5$.

Table 6.3. Illustration of the D-Algorithm for $X/0$ in the Circuit SI .

1	Operation	Gate	A	B	X	Y	W	U	F	G	H	Z
2	Initialization		x	x	x	x	x	x	x	x	x	x
3	PDCF	$G1$	0	0	D							
4	\cap		0	0	D	x	x	x	x	x	x	x
5	D-drive	$G2$			D		1		D			
6	\cap		0	0	D	x	1	x	D	x	x	x
7	D-drive	$G3$						0	D	D'		
8	\cap		0	0	D	x	1	0	D	D'	x	x
9	D-drive	$G5$								D'	0	D'
10	\cap		0	0	D	x	1	0	D	D'	0	D'
11	Justification	$H = 0$				x			0		0	
12	\cap		0	0	D	0	1	0	Ψ	D'	0	D'
13	Justification	$H = 0$				0			x		0	
14	\cap		0	0	D	0	1	0	D	D'	0	D'

As we develop the cubes, the PDCF and the propagation and justification cubes, we list the results as shown in Table 6.3. The first row of the table lists all the nodes and the second column lists the various gates used in propagation and the lines used in justification.

1. Initialize the circuit by placing x on each node (row 2).
2. For a SA0, $X = D$ and hence $A = B = 0$. The PDCF is then $\{0,0,D\}$, as shown in the table in row 3.
3. Now we take the intersection of these two cubes in the two rows according to the operations of Table 6.2 and obtain the cube in row 4.
4. Propagating D through $G2$, we form the D-cube $\{W,X,F\} = \{1,D,D\}$ shown in row 5 and intersect it with row 4. The primary input, W , which used to be indeterminate, is now forced to logic 1. This is consistent with the operations in Table 6.2.

5. Select a sensitizing path from the list of possible sensitization paths. We selected the path through $G3$. Thus our aim is to propagate the D through this path to the primary output, Z .
 - 5.1. To propagate through $G3$, we form the D -cube $\{U,F,G\} = \{0,D,D\}$.
 - 5.2. Again we intersect with the previous cube and obtain the results in row 8.
 - 5.3. The value of Z is still x , so we need to propagate the frontier.
 - 5.4. To propagate through $G5$, we construct the cube $\{G,H,Z\} = \{D',0,D'\}$ given in row 9.
 - 5.5. The intersections with row 8 yield row 10.
6. Since we reached the primary output, it is time to justify the signals on H , Y , U , and W .
7. To justify H , we place the cube $\{F,Y,H\} = \{0,x,0\}$ and intersect to obtain row 10. We run into a conflict since we cannot force F , which carries the fault, to 0. Here we reach an inconsistency and we need to select another cube, if there is one to justify H . We then select another cube $\{x,0,0\}$ and intersect with row 10.

Since we justified successfully all values that we encountered as we were propagating, the set of signals on the primary inputs forms the test pattern: $ABUWY = 00010$. Notice also that every time we construct a cube we need to intersect with the preceding one. We can do the two steps and represent them on the same row. Table 6.3 can thus be reduced to the form shown in Table 6.4. For subsequent examples we use this more compact representation.

Table 6.4 Compaction of Table 6.3.

1	Operation	Gate	A	B	X	Y	W	U	F	G	H	Z
2	Initialization		x	x	x	x	x	x	x	x	x	x
3	PDFC	$G1$	0	0	D	x	x	x	x	x	x	x
4	D frontier	$G2$	0	0	D	x	1	x	D	x	x	x
5	D frontier	$G3$	0	0	D	x	1	0	D	D'	x	x
6	D frontier	$G5$	0	0	D	x	1	0	D	D'	0	D'
7	Justification	$H=0$	0	0	D	x	1	0	Ψ	D'	0	D'
8	Justification	$H=0$	0	0	D	0	1	0	D	D'	0	D'

6.3.2 Case of a Primary Input

The stuck-at-1 fault on Y is selected to illustrate application of the algorithm on a primary input. The results of the process are shown in Table 6.5. For the fault $Y/1$, D' is placed on this line as shown in row 3 of the table. This value is propagated through gate $G4$ and the corresponding D -cube would then be $\{Y,F,H\} = \{D',1,D'\}$. The implication of $F = 1$ is indicated in the fifth row of the table and results in $G = 0$. The propagation is then through $G5$: $\{H,G,Z\} = \{D',0,D'\}$. Since $G = 0$, we need only justify $F = 1$, which requires that $W = X = 1$ and consequently, that $A = B = 0$. The test pattern $ABUWY = 00x10$ is indicated in the last row of the table in boldface. There are therefore two patterns to detect this fault.

Table 6.5. The D-Algorithm for $Y/1$ of the Circuit in Fig. 6.1

1	Operation	Gate	A	B	X	Y	W	U	F	G	H	Z
2	Initialization		x	x	x	x	x	x	x	x	x	x
3	PDCF	Y	x	x	x	D'	x	x	x	x	x	x
4	D frontier	$G4$	x	x	x	D'	x	x	1	x	D'	x
5	Implication	$F=1$	x	x	x	D'	x	x	1	0	D'	x
6	D frontier	$G5$	x	x	x	D'	x	x	1	0	D'	D'
7	Justification	$F=1$	x	x	1	D'	1	x	1	0	D	D'
8	Justification	$X=1$	0	0	1	D'	1	x	1	0	D	D'

6.3.3 Case of a Primary Output

Consider the fault $Z/0$ in circuit $S1$ of Fig. 6.2. It is directly observable and no sensitization is needed, only justification. First, we place a D on this line, but we have several choices for PDCF. It is sufficient to have one of the gate inputs be equal to 1. If we select $G = 1$, H can remain indeterminate and it does not have to be justified. The PDCF is $\{U,H,Z\} = \{1,x,D\}$ and shown in row 2 in Table 6.6. To justify $G = 1$, we need both inputs of $G3$ to be 0. Next, we justify $F = 0$. Here again we have choices: $W = 0$, $X = 0$, or both. Since W is a primary input, it is better to select W , which results in a test pattern: $ABUWY = xx00x$. If, instead, $W = 0$, we have $X = 0$, this value then has to be justified also as shown in rows 5 and 6 or 7. The corresponding patterns would then be $ABUWY = x10xx$ and $1x0xx$, respectively. The two patterns are shown in rows 4th and 6th of Table 6.6. Thus few test patterns are available to detect this fault. If you consider the three patterns and you substitute 0 and 1 for x_5 , you find that there are 14 patterns. These patterns indicate that it is sufficient to specify two of the five inputs to detect the fault. The other

three inputs are unspecified, don't care values. It is advantageous to have some of the inputs uncommitted because when patterns have been generated for all faults, they can be merged in a shorter test. Compacting the test set length by taking advantage of the don't care values is not a trivial matter, and there are static and dynamic methods to perform the compaction.

Table 6.6. The D-Algorithm for Z/0 of the Circuit in Fig. 6.1

1	Operation	Gate	<i>A</i>	<i>B</i>	<i>X</i>	<i>Y</i>	<i>W</i>	<i>U</i>	<i>F</i>	<i>G</i>	<i>H</i>	<i>Z</i>
2	PDCF	<i>G5</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	1	<i>x</i>	<i>D</i>
3	Justification	<i>G</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	0	0	1	<i>x</i>	<i>D</i>
4	Justification	<i>F</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	0	0	0	1	<i>x</i>	<i>D</i>
5	Justification	<i>F</i>	<i>x</i>	<i>x</i>	0	<i>x</i>	<i>x</i>	0	0	1	<i>x</i>	<i>D</i>
6	Justification	<i>X</i>	<i>x</i>	1	0	<i>x</i>	<i>x</i>	0	0	1	<i>x</i>	<i>D</i>

6.3.4 Alternative Strategies

In the examples of Section 6.3.3 consistently propagated to a primary output, stepping through several gate levels in the circuit. This was then followed by justification by stepping backward over the same levels. Whenever we encountered an inconsistency, we backtracked and selected the other *D*-frontier, or another alternative for justification. To accelerate the process, it is possible to follow another strategy. For example, it is possible to propagate the frontier over only one level of logic (one gate), then immediately justify the lines backward for one level (one or several gates depending on the fanin). In this fashion, when inconsistency is encountered, it is handled without having to go through many levels of propagation and justification.

6.4 Critical Path

This algorithm traces a path from a primary output to primary inputs [Abramovici 1984]. In this process, segments of the path are critical. A *critical line* is such that if its logic value is changed, it will change the logic value of the corresponding output. For example, if a line of the path has a value 1 and it changes to 0 because of a stuck at 0 on the line, this fault will be observed at the output. Stuck-at faults of all such critical segments are detected by the same pattern.

The process is similar to justification; it consists of stepping backward on the circuit, one gate at a time. It would help to know how to determine the critical path for elementary gates. Figure 6.5a shows the critical values for simple logic gates: AND, OR and XOR. In all cases the critical values are indicated in bold. For an AND gate, a 1

critical at the output requires that all inputs be 1. If any one input changes to 0, this will be observed on the output of the gate. Thus all inputs are 1 critical. For a 0 critical at the output of the gate, only one of the inputs at a time is set to 0, with all other inputs set to 1. If this input is changed to 1, it will be observable at the output. But if any of the other inputs is changed to 0, the output is not affected.

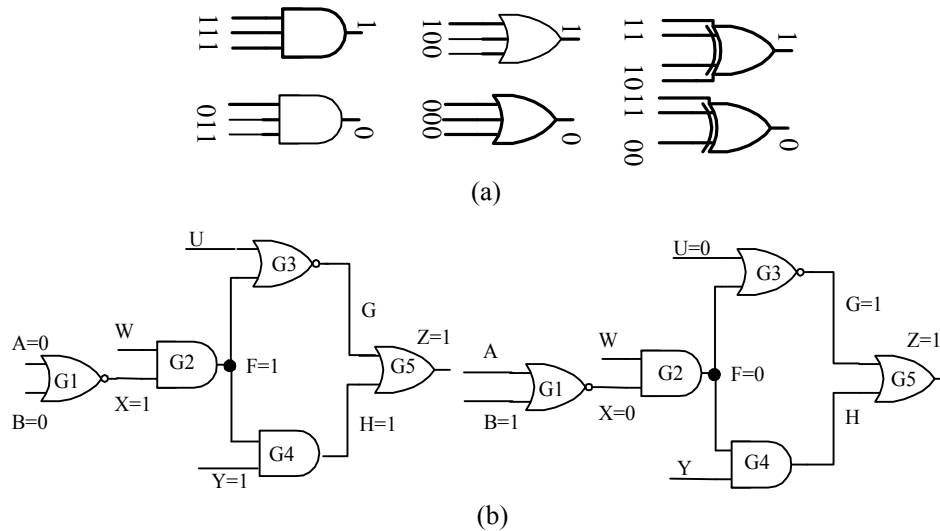


Figure 6.5 Critical Paths: (a) Determination of Critical Paths, (b) Illustration on Circuit S1 in Fig. 6.2, (c) Another Critical Path for S1

For an OR gate, the results are, of course, the dual of the AND gate. A 1 critical on the output of an XOR gate implies an odd number of inputs with 1, and the other inputs are 0. A change on any of the inputs will change the output to 0. Similarly, a 0 critical on the output requires an even number of inputs with 1. Here zero is also considered as an even number. To determine a critical path, we follow four steps listed below:

1. Select a primary output and define 1 (or 0) as the critical value.
2. Justify the critical value to the input of the gate and determine the critical inputs.
3. Repeat step 2 until all primary inputs are determined.
4. If the justification is inconsistent, backtrack and repeat steps 2 and 3.

We apply this algorithm to circuit S1 in Fig. 6.2 and summarize the results in Table 6.7. The headings of the table are the same as those used for the D-algorithm. Let us start with $Z = 1$. For this, it is necessary but also sufficient that either G or H be 1. Let's specify that $H = 1$ and $G = 0$. In this fashion, H is critical since, if it is changed to 0, Z also changes and a fault on H is observed on Z . Had we set both H and G equal to 1, then neither

would be critical since changing one of the them to 0 would not change the output. Since $G4$ is an AND gate, $H = 1$ implies that $Y = F = 1$, and they are all critical. The justification of G is straightforward since it is not critical, but since $F = 1$, there is need to assign 1 to U . Justifying $F = 1$ results in $W = X = 1$ (row 5 in Table 6.7). If $X = 1$, then $A = B = 0$. All critical values in the table are indicated in bold.

Table 6.7. Critical Path: $Z = 0$ in Circuit S1

1	Gate	A	B	X	Y	W	U	F	G	H	Z
2	$G5$	x	x	x	x	x	x	x	0	1	D
3	$G4$				1			1		1	
4	\cap	x	x	x	1	x	x	1	0	1	D
5	$G2$			1		1		1			
6	\cap	x	x	1	1	1	x	1	0	1	D
7	$G1$	0	0	1							
8	\cap	0	0	1	1	1	x	1	0	1	D

The pattern obtained by the steps just described is $ABUWY = 00x11$. The critical values are indicated in bold on the corresponding lines of the schematic in Fig. 6.5b. The faults detected by this pattern are $A/1$, $B/1$, $Y/0$, $X/0$, $F/0$, $W/0$, $H/0$ and $Z/0$.

In addition, we could have selected G instead of H to be critical ($G = 1$ and $H = 0$); then both U and F are critical 0. Justification of F results in a critical 0 for either W , or X . Selecting X is more advantageous since it will, in turn, imply either A or B being critical 1. The more critical segments we have, the more faults are detected by the pattern. The critical path in this case is indicated on the circuit in Fig. 6.5c by the values assigned to the corresponding lines. The test pattern is $ABUWY = 1x0xx$. The detected faults would then be $A/0$, $X/1$, $F/1$, $U/1$, $G/0$ and $Z/0$. In justifying F , we selected $X = 0$. If, instead, we selected $W=0$ and $X=1$, then we would have included $W/1$ instead of $X/1$ in the list of faults detected and the pattern would have been $ABUWY = 0000x$.

6.5 Backtracking and Reconverging Fanout

Both algorithms discussed so far are inefficient when backtracking in a class of circuits that uses reconvergent fanout with XOR gates. We illustrate this with the example shown in Fig. 6.6. Consider the fault $H/0$. The PDCF is $\{A,B,H\} = \{1,1,D\}$. To propagate this frontier to the output, R , we need, for example, $N = Q = 1$. However, this is not possible since N and Q realize complementary functions. Assume that we are not aware of this fact and we

proceed with the justification. We first justify N arbitrarily with $J = 0$ and $K = 1$. Next, we proceed with Q . We can have either $L = M = 1$ or $L = M = 0$, as illustrated in the search tree shown in Fig. 6.7. The implication of the first assignment is that $F = G$ and, consequently, $K = 0$. This is inconsistent with the assumption made earlier in the justification of N . Now we turn to the second case, which requires that F and G are complementary to each other ($K = 1$) as well as to C and E . This implies that $J = 1$, which is inconsistent with the original value of J obtained when N is justified. Tracing back will take us again to node Q , and now we try to propagate the D -frontier using $Q = 0$ instead and obtaining D' at the output. This excessive backtracking motivated [Goel 1981] to develop a new test pattern generation algorithm that is known to run an order of magnitude faster than the D-algorithm [Agrawal 1988].

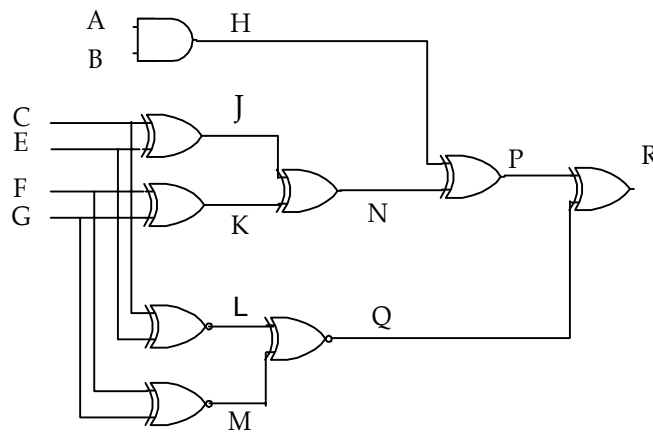


Figure 6.6 Circuit S2: Error Correction and Translation Circuit [Goel 1985]

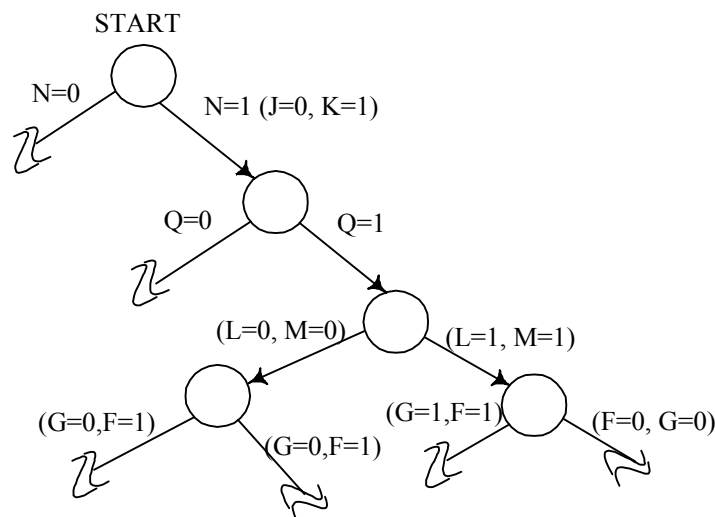


Figure 6.7 D-Algorithm Search Tree for Circuit in Fig. 6.6

6.6 PODEM

The path-oriented decision-making (PODEM) algorithm starts the search for the test pattern at the primary inputs of the circuit. The algorithm is outlined by the flowchart in Fig. 6.8. Starting with an objective, a specific fault to be detected, a search tree is created in which two choices are available, 0 and 1, for the primary inputs. The choice is random. Evaluate the implications of this choice on the subsequent gates to the output. If it furthers the objective -- controlling the fault site to the intended value -- accept it and select another PI. If an inconsistency occurs, the algorithm backtracks and selects another input combination. The search stops whenever a pattern is generated or no patterns are possible (undetectable fault).

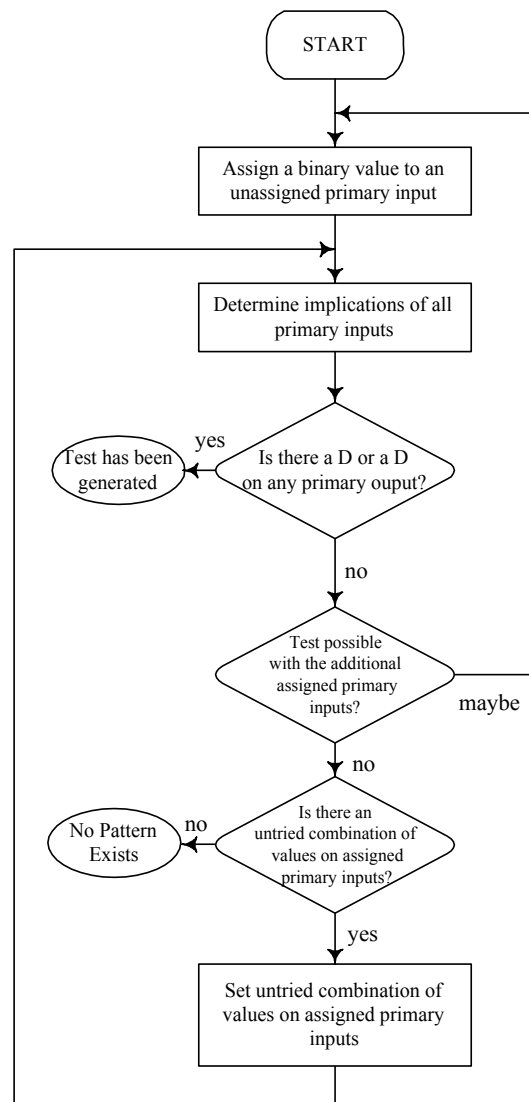


Figure 6.8 PODEM Algorithm

The advantage of PODEM is that it cuts down on the backtracking, as will be demonstrated using the circuit S2 in Fig. 6.6. This is the same circuit that was used to illustrate the difficulty encountered by the D-algorithm with reconvergent fanout. The target fault is $H/0$ and the search tree is shown in Fig. 6.9.

1. Assign x to all inputs.
2. Assigning 0 to the primary input, A , causes $H = 0$; hence this choice does not further our goal and it is discarded. Then $A = 1$.
3. Similarly, $B = 0$ is rejected and $B = 1$ is selected.
4. We proceed with $C = 0$. The implication of this value is not furthering the objective, nor is it blocking it.
5. Thus we select $E = 0$; this then results in $J = 0$ and $L = 1$.
6. Similarly, we select $F = G = 0$, which implies that $K = 0$, $M = 1$, $N = 0$, and $Q = 1$.
7. We can propagate D on P , then D' on the primary output, R .

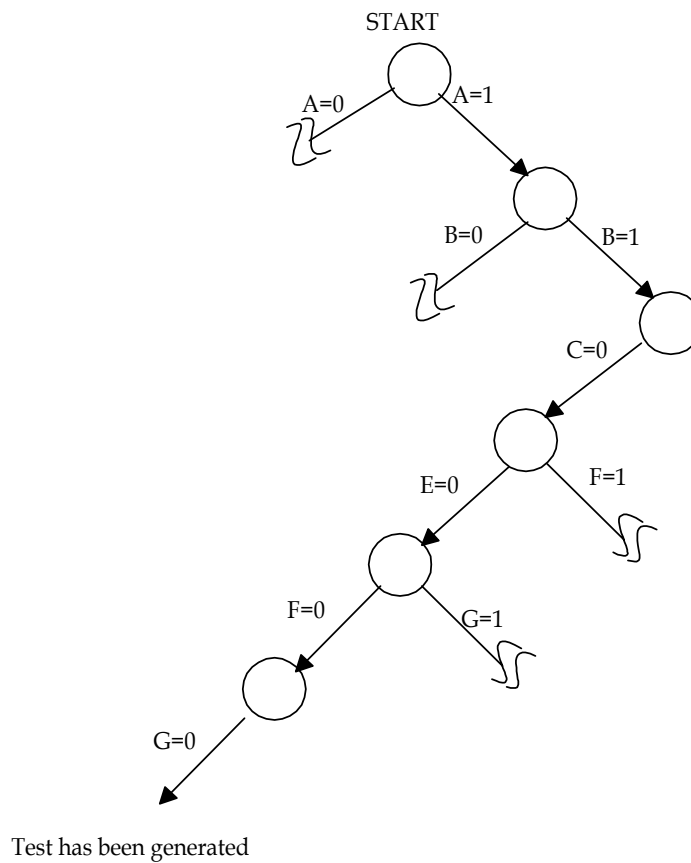


Figure 6.9 PODEM Search Tree for Circuit in Fig. 6.6

From the search tree in Fig. 6.8, the primary inputs are then $\{ABCEFG\} = \{110000\}$. Compared to the D-algorithm, the backtracking is definitely reduced. The algorithm was also applied to two other circuits, as illustrated in Figs. 6.10 and 6.11.

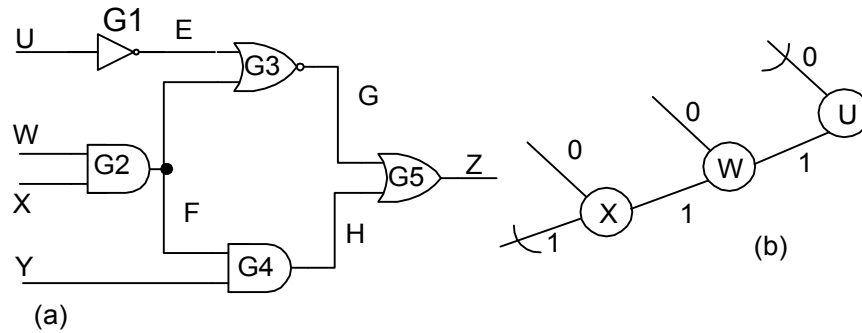


Figure 6.10 Application of PODEM to Circuit S3

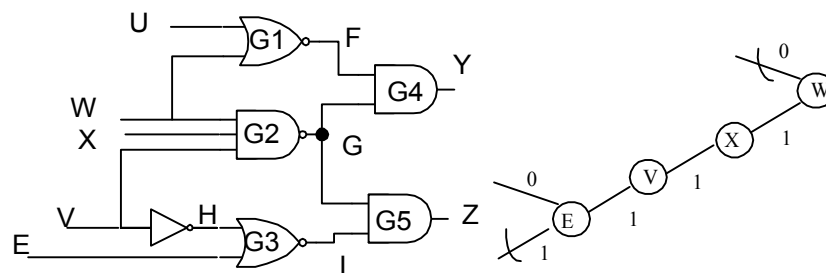


Figure 6.11 illustrating the use PODEM on Circuit S4

6.7 Other Algorithms

PODEM has been effective in reducing the occurrences of backtracking, but two new strategies have helped in reducing them even further: FAN [Fujiwara 1983] and SOCRATES [Schultz 1988]. These algorithms do extensive analysis of the circuit connectivities before backtracking. Also, the search is aided by testability analysis.

6.7.1 FAN Algorithm

The FAN Fan-out-oriented test generation algorithm remedies the exhaustive approach of PODEM by pruning from the search tree any branching that would not yield a solution. FAN uses the following strategies [Fujiwara 1986]:

1. In each step of the algorithm, determine as many signal values as possible that can be implied uniquely.
2. Assign a faulty signal D or D' that is uniquely determined or implied by the fault in question.

3. When the D -frontier consists of a single gate, apply a unique sensitization.
4. Stop the backtrack at a headline, and postpone the line justification for the headline to later.
5. Multiple backtracking is more efficient than backtracking along a single path.
6. In the multiple backtrack, if an objective at a fanout point, p , has a contradictory requirement, stop the backtrack so as to assign a binary value to the fan-out point.

Table 6.8. Comparison of PODEM and FAN Algorithms [Fujiwara 1983].

Circuit	Computing Time		Average Backtracks		%age of Faults Aborted	
	PODEM	FAN	PODEM	FAN	PODEM	FAN
1	1.3	1	4.9	1.2	0.48	0.11
2	3.6	1	42.3	15.2	3.49	1.38
3	5.6	1	61.9	0.6	5.05	0
4	1.9	1	5.0	0.2	0.26	0
5	4.8	1	53.0	23.2	4.79	2.17

Source: [Fujiwara 1983].

Using these strategies, FAN minimized the backtracks and reduced the test generation time as illustrated in a study that compared FAN to other algorithms [Fujiwara 1983]. Only data relevant to FAN and PODEM are presented in Table 6.8 for five benchmark circuits. The last two columns are of particular interest because they indicate the effectiveness of the algorithm. These columns give the number of faults that were not detected after 1000 backtracks; they are referred to as the *aborted faults*.

6.7.2 SOCRATES

The "SOCRATES" (structure-oriented cost-reducing automatic test pattern generation) algorithm is based on FAN, but uses an improved implication procedure and a unique sensitization procedure. In addition, this algorithm reduces backtracking by early recognition of conflicts, and it utilizes several heuristics, which are applied at different states of the test generation process.

6.8 Testing Sequential Circuits

There are several reasons why test pattern generation for sequential circuits is more difficult than for combinational circuits [Hennie 1974, Breuer 1976 and Miczo 1983], among others, have addressed these reasons, which we stated briefly here.

- Most real circuits are sequential in nature.
- The major difficulty in testing sequential circuits is that the output response of the circuit depends not only on the input patterns, but also on the internal states of the circuit. Notice that this sequential circuit may be synchronous or asynchronous and the states are not always directly observable.
- There is a need to drive the circuit first in a known state before applying the test pattern that will sensitize the fault to a primary output. Initializing the circuit might itself require more than one pattern. Therefore, the order in which the test patterns are applied is critical to fault detection. Several techniques have been used to initialize the circuit to a known state before testing. The most forward approach is the use of master set/reset. Another alternative is the use of a synchronizing sequence that forces the circuit in a final state regardless of its initial state [Hennie 1968]. However, not all circuits have synchronizing sequences. Also, under faulty conditions, there is no guarantee that either initialization technique will work.
- Timing is another factor that complicates test pattern generation. For proper operation of sequential circuits, adequate consideration needs to be given to setup and hold times. If timing restrictions are not followed, the circuit may behave in an unpredictable manner. In addition, even when the test is applied according to the timing specification, it is possible that the delays of the different components of the circuit produce *hazard* and race conditions. Hazards can exist in both combinational and sequential circuits because of component delays (logic hazard) or as an inherent function of the circuit logic itself (functional hazard). Whereas for a combinational circuit, the signal will eventually end in its correct value before strobing, this may not be the case for sequential circuits. A hazard condition may put the circuit in a faulty state. For example, in a master-slave flip-flop, the slave may capture the static hazard on the master. This phenomenon, known as *essential hazard* [McCluskey 1986], can prevent detection of the fault for which the test was intended.

In the remainder of this section, we examine sequential test pattern generation using a functional approach and a fault-oriented deterministic method that has been adapted to sequential circuits.

6.8.1 Functional Testing

It is possible to test a sequential circuit by applying a specific input sequence that exercises its function. However, such a test does not guarantee a complete stuck-at fault testing of the circuit unless simulation is used. Another

approach is to verify the operations of the circuit in accordance with its state table [Moore 1956, Hennie 1968]. This method requires the enumeration of all possible fault machines and yields an experiment that not only detects faults, but also identifies which fault has occurred. The methodology is to sequential circuits what exhaustive testing is to combinational circuits. Although it is not practical for large FSMs, it gives an insight into the validation of FSMs, as we find out next.

6.8.1.1 Checking Experiment

A fault will transform an FSM, M , into some other state, M_f . Assuming that the number of states has not increased, the checking sequence will distinguish M from M_f . In general, only the primary inputs can be controlled and only primary outputs can be observed. We have seen that for a completely specified FSM, the mathematical formulation gives a unique next state and an output for each transition. The states are not observable directly, but they can be recognized from the output sequences.

To verify the correctness of an FSM, we apply a specific input sequence and we observe the output sequence. For the fault-free machine, the sequence initializes M into a known state, then forces it to pass through all possible transitions. The input sequence consists of three main sequences: a *synchronizing* or *homing sequence*, a *distinguishing sequence*, and a *transition sequence*.

- The synchronizing sequence (SS) places the M in a known state.
- The homing sequence (HS) places M in a known state that is identifiable from the output response sequence.
- The distinguishing sequence (DS) produces an output sequence that defines uniquely the initial state of M at which the sequence was applied.
- The transition sequence (TS) indicates the transition from one state to any of the other states.

Table 6.9 State Table for FSM M

Present State	Next State	
	$I=0$	$I=1$
A	$C,1$	$B,0$
B	$C,0$	$B,1$
C	$D,1$	$C,1$
D	$A,1$	$C,0$

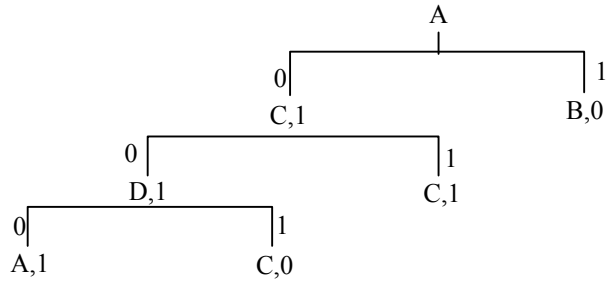


Figure 6.12 The Transition Sequence for FSM M

Let us illustrate the development of these sequences using machine M , as defined algebraically in Section 3.2.3 and the state table form that is shown in Table 3.3 and reproduced in Table 6.9. Finding the transition sequence is very easily developed from either form. In passing from state A to any of the other states, B , C and D , we need to apply the sequences 1, 0, and 01 as indicated on the binary tree in Fig. 6.12. In this representation, the line segments labeled 0 and 1 represent the transitions from one state to another for input 0 and 1. We identify the transition from A to any of the other states by the unique output response shown in the table.

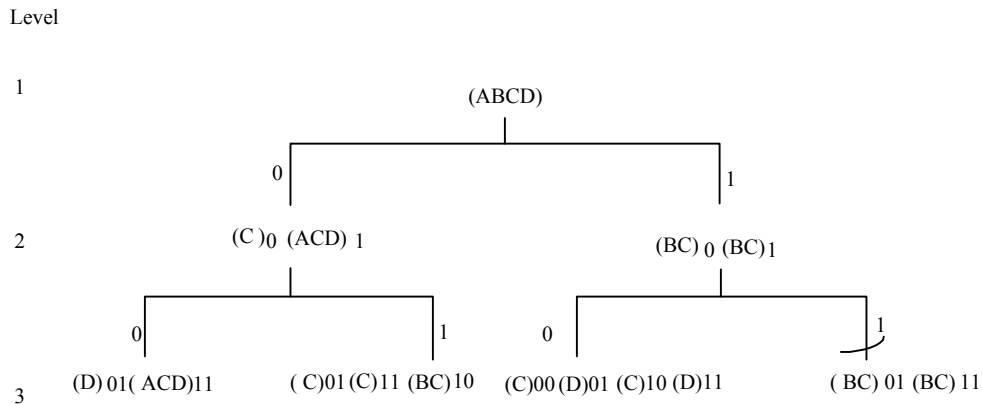


Figure 6.13 The Distinguishing Sequence for FSM M

To develop the other sequences, we use a *successor tree*. Initially, M may be in any of the four states $\{ABCD\}$. We are not certain of the exact state and we call this collection of states the initial *uncertainty* of the machine [Kohavi 1978]. If we apply to M input 0, the next states will be either (C) with output 0 or any of the states (ACD) with output 1. (C) and (ACD) are the *zero-successors* of the initial uncertainty $\{ABCD\}$ and they form an *uncertainty vector*. Similarly, we can generate the 1-successors by applying 1 to $\{ABCD\}$. We then obtain the 1-successors, which are (BC) with 0 and (BC) with 1. To each uncertainty, we again apply input 0 and 1 and generate higher-level uncertainties. The formation of these uncertainties is shown in Fig. 6.13. The subscripts indicate the

output sequence for the machine transitions. The process of building the tree is terminated unsuccessfully as we obtain an uncertainty that has already been obtained. In this example, the rightmost branch with 11 terminates in $(BC)_{11}$ and need not be explored any further. We elaborate later on successful termination conditions, which depend on the sequence sought, homing or distinguishing.

Table 6.10 (a) Homing Sequences, (b) Application Distinguishing Sequences 10.

HS	Final State	Output
0	C	0
00	D	01
01	B	01
	C	11
10	C	00
	D	01

(a)

Distinguishing Sequence		
Initial State	Final State	Output sequence
A	C	00
B	C	10
C	D	11
D	D	01

(b)

The vectors that have components with single states are called *trivial uncertainties*; those that consist of single states and identical states are called *homogeneous uncertainties*. For example, $(BB)CD$ is a homogeneous uncertainty, whereas $(A)(B)(D)$ is a trivial uncertainty.

For the *homing sequence*, the process is terminated successfully when a trivial or homogeneous vector is obtained because at least one single state is identified. The tree in Fig. 6.13 shows four different homing sequences, which are listed in Table 6.10a. The first sequence is of length 1 and the others are of length 2. Any of these sequences will guarantee the final state with a unique output sequence, but it does not identify the initial state. Notice that the homing sequence does not distinguish the initial states.

Once we initialize the FSM, we can apply the *distinguishing sequence* to verify the correctness of the machine. For the distinguishing sequence, we follow the same procedures as for the homing sequence except that we terminate the process when we obtain a trivial uncertainty vector. If we reach a homogeneous uncertainty, we abandon the search along this path. For machine M , there is one distinguishing sequence, which is 10, one of the homing sequences. A distinguishing sequence is a homing sequence, but not vice versa. The response of the machine to this *DS* sequence is summarized in Table 6.10b. The output strings distinguish the initial states uniquely.

The initialization of the FSM may also be accomplished using the synchronizing sequence. This sequence is developed using a successor tree; however, this time, there is no need to separate the uncertainties by their output. A path in the tree is terminated when a repeated uncertainty is obtained and terminates successfully when a trivial uncertainty with a singleton state is obtained. Usually, this SS is used to initialize the FSM. However, the existence of such a sequence cannot be guaranteed. The successor tree, which yields the synchronizing sequence 101, is shown in Fig. 6.14; it will put M in state C . This sequence is 3 bits long, whereas the homing sequences are at most 2 bits long. In general, it is more beneficial to use shorter sequences.

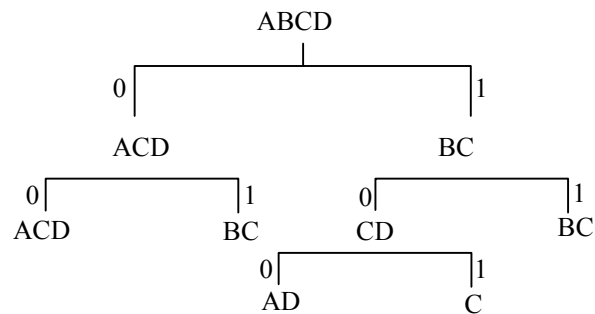


Figure 6.14 The Synchronizing Sequence for FSM M

Next we use the various sequences generated for M to verify the correctness of the machine:

1. Initialize the M using an SS or HS to state S_0 .
2. Apply a DS to verify this state.
3. Let the final state obtained in the preceding step be S_i .
4. Apply a DS to verify S_{ic} .
5. Repeat steps 3 and 4 until all states have been identified.
6. If in this process, a state S_j is not reachable, apply a TS to get you to this state and apply a DS to verify it.
7. Use the TS to verify all transitions except for those already verified in step 5.

Applying this process on FSM M , we obtain the checking sequence given in Table 6.11. This table is organized in time steps. For each step, an input is applied on the state in the same column. The next state and the corresponding output are then entered in the next time-step column. Initially, at time 0, the FSM is represented by its total uncertainty $\{ABCD\}$ and the output is not defined. Then 0 is applied on the input to home the machine to state C . Thus C and its corresponding output, 0, are entered for the state and the output in time step 1. Now, to verify state C , we apply the DS (10) in time steps 1 and 2 and observe the output sequence $\{11\}$, which according to Table 6.10b, is the signature of state C for the distinguishing sequence used. The interest in the use of checking experiment

has continued through the 1980's when it has been recommended in conjunction with Built-in self-test (BIST). Since the checking sequence is usually very long, schemes have been proposed to compact the circuit's response to the sequence. We will describe such schemes in Chapter 11.

Table 6.11 The Checking Sequence for the FSM M .

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Input	0	1	0	1	0	0	1	0	0	0	1	1	0	1	0	1	0	0	0	1	1	1	0		
State	$ABCD$	C	C	D	C	D	A	B	C	D	A	B	B	C	C	D	C	D	A	B	A	B	B	C	
Output	--	0	1	1	0	1	1	0	0	1	1	0	1	0	1	1	0	1	1						

6.8.1.2 Exhaustive Testing

As for the combinational parts, it is possible to use exhaustive and pseudoexhaustive testing for sequential circuits. The advantages of these types of test pattern generation are simplicity and low cost. However, for an S -state FSM with N primary inputs, an exhaustive test is 2^{S+N} long. A pseudoexhaustive approach in which exhaustive patterns are created for partitions of the circuits was recommended to reduce the test length and proved to be successful for combinational circuits [McCluskey 1981]. This approach was also applied to sequential circuits that include DFT constructs [Wunderlich 1989]. Because exhaustive test patterns do not use the functionality of the circuits, they are extremely long.

6.8.2 Deterministic Test Pattern Generation

Very few algorithmic techniques exist for testing sequential circuits. Some of the methods have been devised for asynchronous circuits but can be adjusted for use with synchronous circuits. For the latter type, we use the Huffman model for FSM that has been described in Chapter 3 [Huffman 1954]. The circuit consists of a combinational part and a set of flip-flops as illustrated in Fig. 6.15a. For simplicity, only D flip-flops are used since it is not difficult in any case to transform other flip-flops to this type.

[Seshu 1965] developed the first algorithm for testing sequential circuits. This technique, which predates the D-algorithm, does not require state tables as it is simulation-based. It starts with a vector and simulates both the good and faulty machines. Then the input sequence is changed one bit at a time, and the effectiveness of the changes is evaluated to select the change that detects most of the faults. The effectiveness of the patterns was determined according to four heuristics that are described concisely by [Miczo 1986].

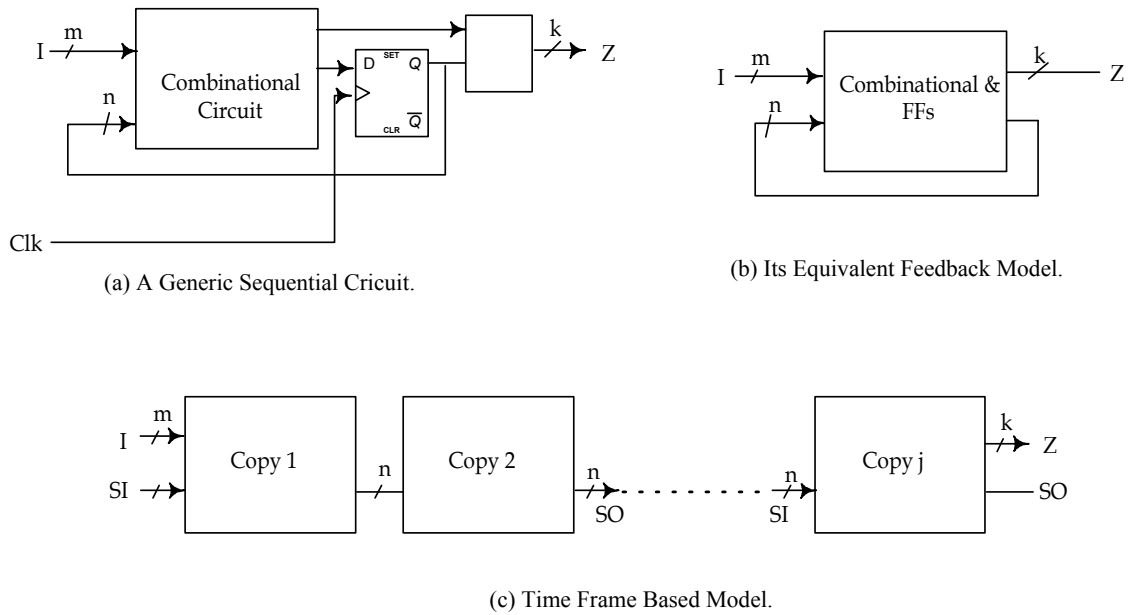


Figure 6.15 Sequential Test Pattern Generation: Iterative Logic Array.

A heuristic bearing resemblance to Seshu's consists of cutting all feedback loops of a sequential circuit and thus transforming it into a combinational circuit [Kubo 1968]. It is assumed that all flip-flops in the circuit are fault-free. Test patterns are generated for the transformed circuit using the D-algorithm. The technique is applied only to synchronous machines and, in this manner, race and hazard problems are avoided. Once a test pattern is generated, it is important to check that the feedback lines have the same value at the inputs and the outputs. If this is the case, a test pattern has been generated; otherwise, another pattern needs to be generated.

[Putzolu 1971] proposed the iterative test generation (ITG) heuristic. This technique extends and improves Kubo's work. It was intended for asynchronous circuits, but it is also applicable to synchronous circuits. In addition, it does not require that the storage devices be fault-free. However, as a heuristic, it does not guarantee finding a test for the circuit. The ITG algorithm consists of the following steps:

1. Cut the feedback loops according to a heuristic that assigns weights to the different feedback lines and then determines those lines that would be cut.
2. Use the D-algorithm to generate patterns.
3. Run a simulation to check the effectiveness of the test sequence to detect the fault.

The cutting strategy is such that a minimal number of feedback lines are selected to make the circuit acyclic. Each feedback line is then represented by input and output to the circuit. They are labeled the pseudo inputs (*SI's*) and pseudo-outputs (*SO's*). The resulting circuit shows the behavior of the original circuit at a given moment

in time. To examine the behavior in subsequent moments, different copies of the original circuit are produced as illustrated in Fig. 6.15c. The copies are snapshots at different time intervals. If the *SO*'s of one copy are connected to the corresponding *SI*'s of the next copy by a delay, the modified circuit should behave as the original circuit. The D – algorithm is applied to the modified circuit with the following restrictions:

- The fault is detected when it is propagated to a primary output.
- The test patterns cannot be allowed to depend on any particular values of the *SI*. That is, the intersection of x and any other value is always ϕ on the *SI*.

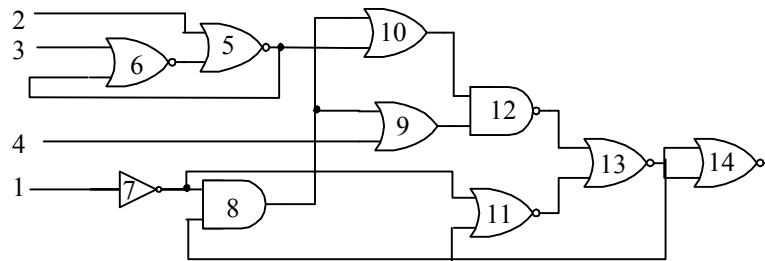


Figure 6-16 Circuit S5

We illustrate the method by generating the pattern to detect the SA1 fault on the output of gate 7 (G7) of the circuit *S5* shown in Fig. 6.16. This circuit is the example used in [Putzolu 1971] and in which all NOR gates with their inputs tied together are replaced by inverters. Two time frames are used as illustrated in Fig. 6.17. The second frame is used to place the PDCF and propagate the fault to a primary output. A justification that requires nodes with feedback to have other values than those assigned by the propagation are then placed on the previous frame, frame 1. In explaining the test generation, we refer to a line that is the output of a gate by the number of this gate.

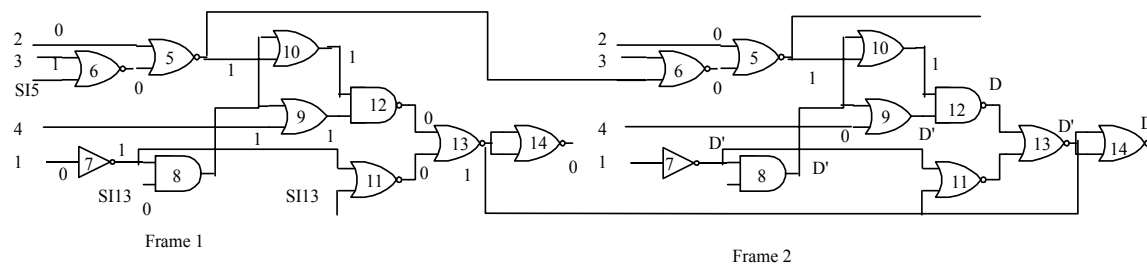


Figure 6-17 Time Frames for Circuit S5

The PDCF is propagated through G11, then G13, and finally to the primary output, G14, as indicated on frame 2 of Fig. 6.17. This propagation requires that 0 be placed on nodes 12 and 13. This terminates the *D*-drive. We then

move to justification on frame 1. A 0 on 13 is obtained by placing 1 on 11 or on 12. A 1 on 11 implies a zero on $PI13$. This is an inconsistency per the second restriction stated above. We attempt next to place 1 on 12, which implies a 0 on 9 or on 10. Any of the two possibilities causes the output of $G8$ to be 0. However, a 0 on 8 requires that a 0 be on either $PI13$ or on 7. Both assignments cause inconsistencies since $PI13$ is pseudoinput and 7 is stuck at 1 and should be held high. Therefore, we need to backtrack and find another D -drive.

Another sensitization path for the fault to the primary output 14 is through $G8$, $G9$ (or $G10$), $G12$, and $G13$ as indicated on frame 2 of Fig. 6.17. This propagation requires 1 on the other input of $G8$, 0 on 4, 1 on 10, and 0 on 11. We move next to frame 1 and justify these values. To get a 1 on 13, we need 0 on 11 and on 12. To realize this value on 11, a 1 is placed on 7, which implies a 0 on 1; for 12, on the other hand, both 9 and 10 should be 1. This requirement implies that a 1 be put on 4 and 5. Consequently, we need a 0 on 2 and a 1 on 3. Thus the justification of 13, 11, and 4 is complete and the pattern is $\{0011\}$. The implication of this pattern and, in particular, having 1 on 5, is placing a 0 on 6 in frame 2. Finally, we justify 1 on 10 by placing 1 on 5. But, since the value of 6 is 1, we need to place 0 on 2. The second pattern becomes $\{10x0\}$. The test sequence is then extracted from the values assigned to the primary inputs in the two frames: 0011 and 10x0.

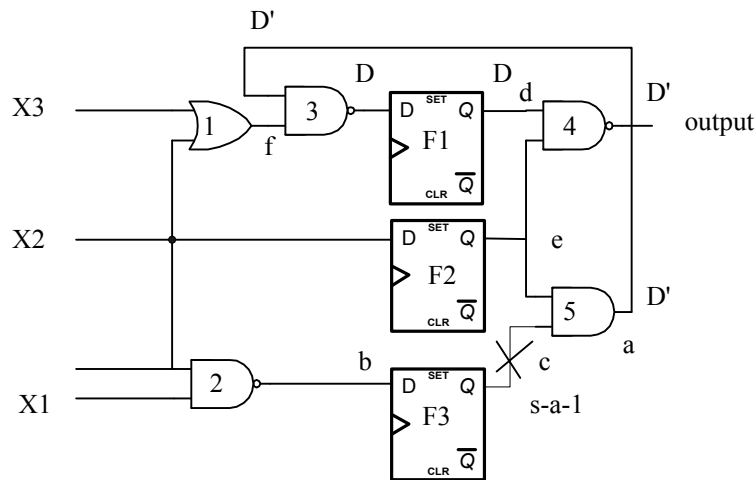


Figure 6.18 Synchronous Circuit S6

The approach is also applicable to synchronous circuits, as we illustrate on the FSM shown in Fig. 6.18. The path from the fault, $c/1$, to the primary output, Z , is through *gates* $G5$, $G3$, $F1$, and $G4$. This requires clocking once while keeping e and f equal to 1. However, we first need to provoke the fault. Therefore, we place a 1 on b and c , and

clock once. We clock a second time to propagate the fault. This is equivalent to two time frames, as illustrated in Fig. 6.19. The test sequence is then $\{x11, x1x\}$. In this case, it is sufficient to use the first pattern and clock twice.

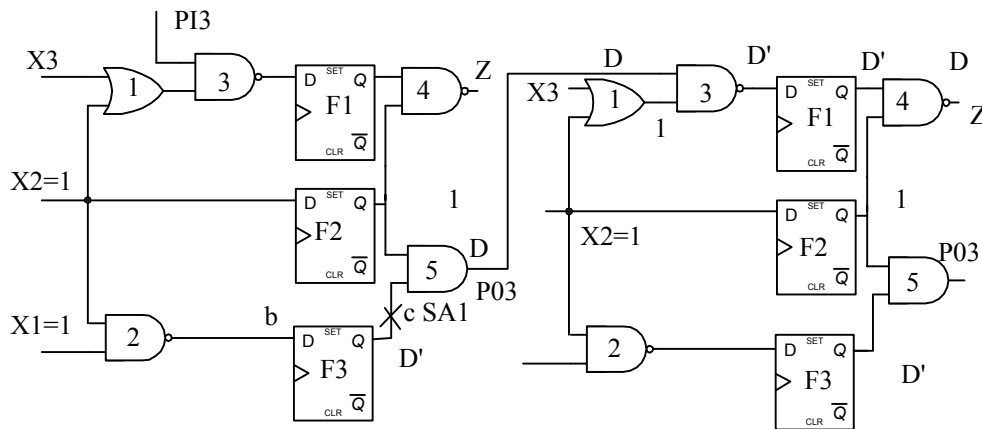


Figure 6.19 Time Frames of sequential Circuit S6

With the advent of scan-path design in early 1970s, the interest in sequential test patterns generation has slowed down since this design technique has reduced the problem to testing combinational circuits. This will be discussed in Chapter 9. Only few algorithms were proposed [Mallela 1985, Marlett 1986]. As the use of scan design became widespread, and synthesis for testability (Chapter 14) started to become popular, the emphasis was on modifying sequential circuits to make them easily testable.

References

- Abramovici, M. et al. (1984), Critical Path Tracing - An Alternative to Fault Simulation, *IEEE Des. & Test of Comput.*, Vol. 1, No. 1., pp. 83 – 93.
- Agrawal, V. D. and S. C. Seth (1988), *Test Generation for VLSI Chips*, IEEE Computer Society Press, Los Alamitos, CA.
- Breuer, M. A. and A. D. Friedman (1976), *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, New York.
- Fujiwara, H. and S. Toida (1982), The complexity of Fault Detection for Combinational Logic Circuits," *IEEE Trans. on Comput.*, Vol. C-31, No. 6, pp. 555 – 560.
- Fujiwara, H. and T. Shimono (1983), On the acceleration of test generation algorithms, *IEEE Trans. on Comput.*, Vol. C-32 No. 12, pp. 1137 – 1144.
- Fujiwara, H. (1986), *Logic Testing and Design for Testability*, MIT Press, Cambridge, MA.
- Goel, P. (1980), Test generation cost analysis and projections, *Proc. 17th Design Automation Conf.*, pp. 77-84.

- Goel, P. (1981), An Implicit enumeration algorithm to generate tests for combinational logic circuits, *IEEE Trans. on Comput.*, Vol. C-30 No. 3, pp. 215 – 222.
- Hennie, F. C. (1968), *Finite-State Models for Logic Machines*, Wiley, New York.
- Hennie, F. C. (1974), Fault detection experiments for sequential circuits, *Proc. 5th Symposium on Switching Theory and Logical Design*, pp. 95 – 110.
- Hsiao, M. Y. and D. K. Chia (1971), Boolean Difference for Fault Detection in Asynchronous Sequential Machines, *IEEE Trans. on Comput.*, Vol. C-20, No. 11, pp.1356 –1361.
- Huffman, D. A. (1954), The Synthesis of Sequential Circuits, *J. Franklin Inst.*, Vol. 257, No. V, pp.161 – 190, 273 – 303.
- Ibarra, G. H. and S. K. Sahni (1975), Polynomially complete fault detection problems, *IEEE Trans. on Comput.* Vol. C – 24, No. 3, pp. 242 – 249.
- Kohavi, Z. (1978), *Switching and Finite Automata Theory*, 2nd Ed., McGrawHill, New York.
- Kubo, H. (1968), A procedure for generating test sequences to detect sequential circuit failures, *NEC Res. Dev.*, Vol. 12, No., pp. 69 – 78.
- Larabee, T. (1989), Efficient generation of test patterns using Boolean difference, *Proc. IEEE International Test Conference*, pp. 795 – 801.
- Mallela, S. and S. Wu (1985), A sequential circuit test generation system, *Proc. IEEE International Test Conference.*, pp. 57 – 61.
- Marlett, R. An effective test generation system for sequential circuits, *Proc. 23rd Design Automation Conference*, pp. 250 – 256.
- McCluskey, E.J. and S. Bozorgui-Nesbat (1981), Design for autonomous test, *IEEE Tans. Comput.*, Vol. C-30, No. 11, pp. 860 – 875.
- McCluskey E.J. (1986), *Principles of Logic Design*, Prentice Hall, Upper Saddle River, NJ.
- Miczo, A. (1983), The sequential ATPG: A Theoretical limit, *Proc. IEEE International Test Conference*, pp. 143 – 147.
- Miczo, A. (1986), *Digital Logic Testing and Simulation*, Wiley, New York.
- Moore, E. F. (1956), Gedanken experiments on sequential machines, *In Automata Studies*, C.E. Shannon and J. McCarthy, (Eds.), Princeton University Press, Princeton, NJ.
- Putzolu, G. and J. P. Roth, (1971), A heuristic algorithm for the testing of Asynchronous Circuits, *IEEE Trans. Comput.*, Vol. C-20, No. 6, pp. 639 – 647.
- Roth, J. P., (1966), Diagnosis of Automata Failures: A calculus and a method, *IBM J. Res. Dev.*, Vol. 10, No pp. 278 – 281.
- Roth, J. P., W. G. Bouricius, and P. R. Schneider (1967), Programmed Algorithms to compute Tests to detect and distinguish between failures in Logic circuits, *IEEE Trans. on Electron. Comput.*, Vol. EC-16 No. 10, pp. 567 – 580.
- Schultz, M. H. et al. (1988), SOCRATES: A Highly Efficient Automatic Test Pattern Generation System, *IEEE Trans. on Comput. Aided Des.*, Vol. CAD-8 No.1, pp. 126 – 137.

Sellers, F. F. et al. (1968), Analyzing errors with Boolean difference, *IEEE Trans. on Comput.*, Vol. C-17, No. 7, pp. 676 – 683.

Seshu, S. (1965), On an improved diagnosis program, *IEEE Trans. on Electron. Comput.*, Vol. EC-14, No. 2, pp. 76 – 79.

Wunderlich, H. J. and S. Hellebrand (1989), The pseudo-exhaustive test of sequential circuits, *Proc. IEEE International Test Conference*, pp. 19 – 27.

Problems

- 6.1. Collapse the faults on the circuit in Fig. P6.1. Then use the D-algorithm to generate test patterns for the circuit. If possible, use a fault simulator to assess the fault coverage of the test set.
- 6.2. Use Boolean difference to detect faults on the primary inputs of the circuit in Fig. P6.1 and compare the results with test patterns generated for the same faults in Problem 6.1.
- 6.3. Use PODEM algorithm to generate test patterns for the circuit in Fig. P6.3.
- 6.4. Determine test patterns that detect the SA0 faults on lines *f* and *b* of circuit S6 in Fig. 6.18.
- 6.5. For the state machine listed in Table P6.5, determine,
 - a) a synchronizing sequence, if any,
 - b) a homing sequence; and,
 - c) a distinguishing sequence.
 - d) develop a checking sequence.
 - e)
- 6.6. Use the state table that you developed for Problem 3.4 and determine its synchronizing sequence.

able P6.5

Pres State	Next State,	
	I=0	I=1
A	C,0	A,1
B	A,1	B,0
C	D,0	B,1
D	B,1	D,0

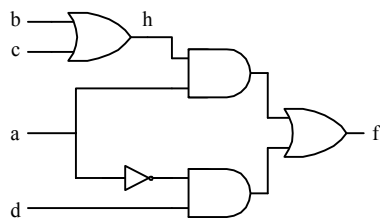


Figure P6.1

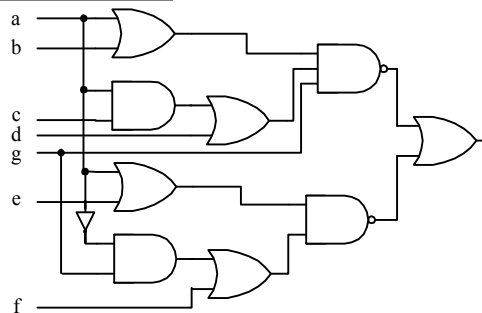


Figure P6.3