1 Introduction to Logic Testing

The development of computers has been stimulated greatly by integrated-circuit technology. Circuit density has increased dramatically, and the cost of devices has decreased as their performance has improved. With these developments, reliability has become increasingly important. However, with the advent of very-large-scale integration (VLSI), testing has come up against a wall of "intractability" and is at a standstill. Since the gate density of VLSI circuits is increasing much more rapidly than the number of access terminals, the ability to generate test patterns and to process fault simulations is deteriorating. The difficulty can be reduced by the development of faster and more efficient algorithms for test-pattern generation or by the use of design techniques to enhance testability.

1.1 Logic Circuits

Logic circuits are constructed by interconnecting elements called *gates* whose inputs and outputs represent only the values denoted by 0 and 1. Some common gates are AND, OR, NOT, NAND, NOR, and EOR (Exclusive-OR); their symbols are shown in figure 1.1. The output of each gate can be represented by a logic function or a Boolean function of the inputs. The terms *logic* and *Boolean* are often used to denote the same meaning. A logic function can be specified by a truth table, a Karnaugh map, or a set of cubes. Figure 1.2 demonstrates these three forms. The Boolean (logic) operations $\cdot$, $+$, and $\bar{}$ correspond to AND, OR, and NOT, respectively.

The output z of an AND gate with inputs x_1 and x_2 is 1 if and only if both of its inputs are 1 simultaneously, and can be expressed as

$$z = x_1 \cdot x_2.$$

The output z of an OR gate with inputs x_1 and x_2 is 1 if and only if any of its inputs are 1, and can be expressed as

$$z = x_1 + x_2.$$

The output z of a NOT gate or an inverter with input x is 1 if and only if its input is 0, and can be expressed as

$$z = \bar{x}.$$

The output z of a NAND gate with inputs x_1 and x_2 is 1 if and only if any of its inputs are 0, and can be expressed as

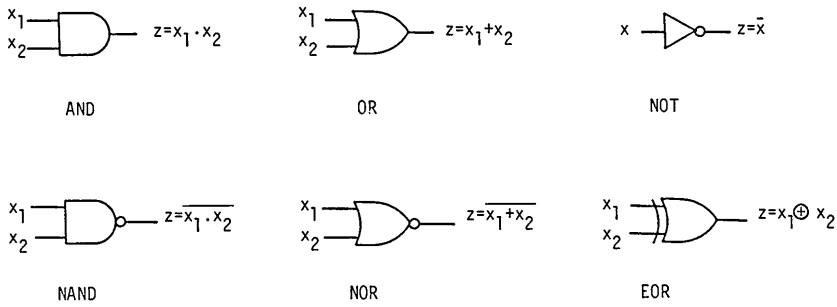
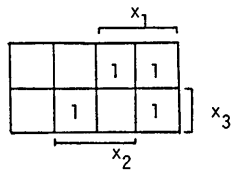


Figure 1.1
Symbols for gates

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

(a) Truth table



(b) Karnaugh map

x_1	x_2	x_3
1	0	
1	-	0
0	1	1

(c) A set of cubes

Figure 1.2
Representations for a logic function

$z = \overline{x_1 \cdot x_2} = \bar{x}_1 + \bar{x}_2$.

The output z of a NOR gate with inputs x_1 and x_2 is 1 if and only if both of its inputs are 0, and can be expressed as

$z = \overline{x_1 + x_2} = \bar{x}_1 \cdot \bar{x}_2$.

The output z of an EOR gate with inputs x_1 and x_2 is 1 if and only if its input are not simultaneously equal, and can be expressed as

$z = \bar{x}_1 x_2 + x_1 \bar{x}_2 = x_1 \oplus x_2$.

Logic circuits can be categorized as *combinational* or *sequential.* A combinational (logic) circuit consists of an interconnected set of gates with no feedback loops. (A feedback loop is a directed path from the output of some gate G to an input of gate G.) The output values of a combinational circuit at a given time depend only on the present inputs. Hence, each output can be specified by a logic function of its input variables. A block diagram for combinational circuits is shown in figure 1.3, where $x_1, \ldots, x_n$ are the inputs and $z_1, \ldots, z_m$ are the outputs.

A sequential (logic) circuit contains feedback loops. The output values at a given time depend not only on the present inputs but also on inputs applied previously. The history of previous inputs is summarized in the *state* of the circuit. Figure 1.4 shows a block diagram for sequential circuits. A sequential circuit consists of two sections: a combinational circuit and a memory circuit. The circuit of figure 1.4 has n primary inputs $x_1, \ldots, x_n$; m primary outputs $z_1, \ldots, z_m$; p feedback inputs $y_1, \ldots, y_p$; and p feedback outputs $Y_1, \ldots, Y_p$. $Y_1, \ldots, Y_p$ are also inputs to the memory, and $y_1, \ldots, y_p$ are outputs of the memory. The *present state* of the circuit is represented by the variables $y_1, \ldots, y_p$, and the *next state* is determined by $Y_1, \ldots, Y_p$.

The mathematical model of a sequential circuit is called a *sequential machine* or a *finite-state machine.* A sequential machine M is characterized by the following:

a finite set S of states,
a finite set I of input symbols,
a finite set O of output symbols,
a mapping N of $S \times I$ into S (called the *next-state function*), and
a mapping Z of $S \times I$ into O (called the *output function*).

The sequential machine M is expressed by the 5-tuple (S, I, O, N, Z).

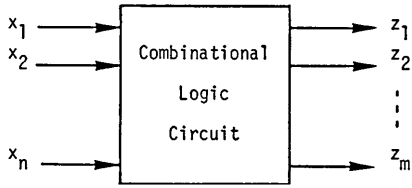


Figure 1.3
Block diagram of combinational logic circuits

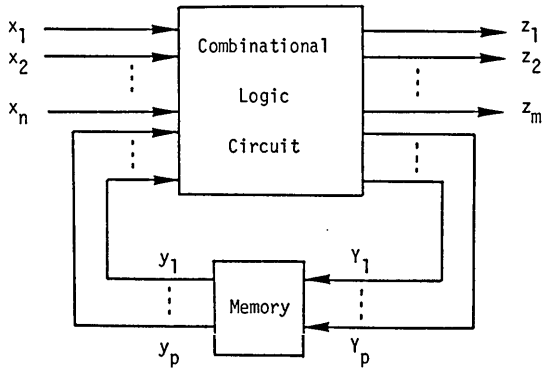


Figure 1.4
Block diagram of sequential logic circuits

A sequential machine can be conveniently represented either in tabular form by a *state table* (*flow table*) or in graph form by a *state diagram*. A state table, as illustrated in figure 1.5(a), lists the present states as row headings and the input symbols (also called input values or input states) as column headings. The entry in row S_i and column I_j represents the next state $N(S_i, I_j)$ and the output $Z(S_i, I_j)$. The machine shown in figure 1.5(a) has four states (labeled 1, 2, 3, 4) and a binary input and output. Figure 1.5(b) shows the state diagram corresponding to the table of figure 1.5(a). A state diagram is a directed graph whose nodes correspond to states of the machine and whose arcs correspond to state transitions. Each arc is labeled with the input value separated by a slash from the output value associated with the transition.

Sequential circuits are categorized as either *synchronous* or *asynchronous*, depending upon whether or not the behavior of the circuit is clocked at

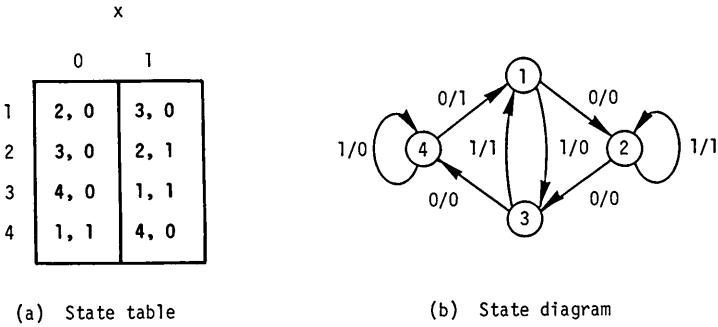


Figure 1.5
Representations for a sequential machine

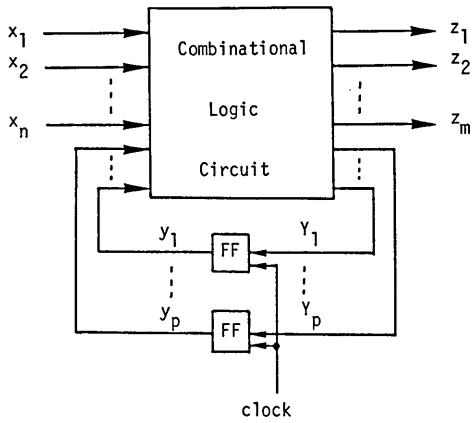
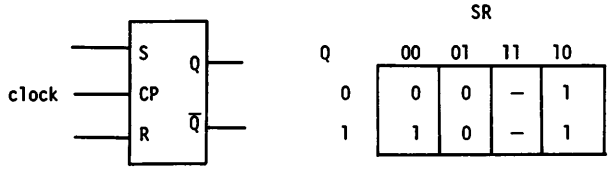
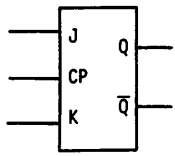


Figure 1.6
Block diagram for a synchronous sequential logic circuit



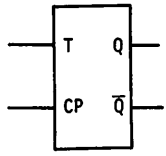
Q	SR 00	01	11	10
0	0	0	–	1
1	1	0	–	1

(a) SR flip-flop



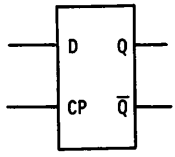
Q	JK 00	01	11	10
0	0	0	1	1
1	1	0	0	1

(b) JK flip-flop



Q	T 0	1
0	0	1
1	1	0

(c) T flip-flop



Q	D 0	1
0	0	1
1	0	1

(d) D flip-flop

Figure 1.7
Representations for flip-flops

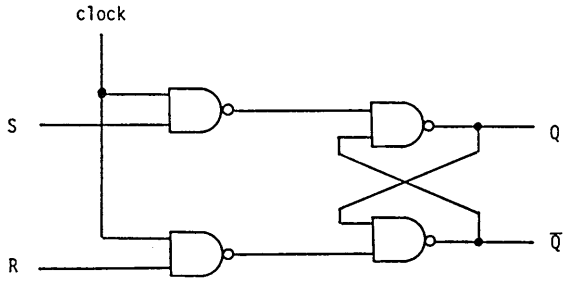


Figure 1.8
Realization of the SR flip-flop

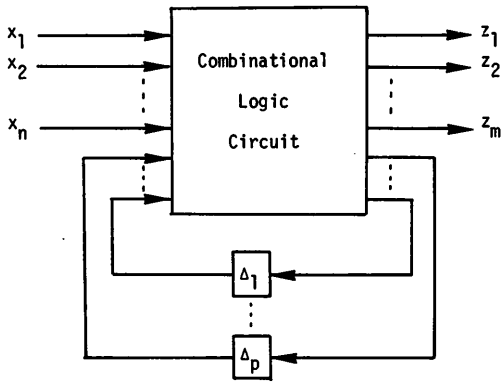


Figure 1.9
Block diagram for asynchronous sequential logic circuit

discrete instants of time. The operation of synchronous sequential circuits is controlled by a synchronizing pulse signal called a *clock pulse* or simply a *clock*. The clock is usually applied to the memory portion of the circuit. Figure 1.6 is a block diagram for synchronous circuits. A series of bistable memory elements called *clocked flip-flops* (FF) are used in synchronous circuits. The most popular memory elements are the D (Delay), T (Trigger), SR (Set-Reset), and JK flip-flops shown in figure 1.7. Figure 1.8 shows a realization of the SR flip-flop. Other flip-flops can be similarly constructed from cross-coupled NAND or NOR gates.

The behavior of an asynchronous circuit is not synchronized; it is unclocked. Each feedback line is assumed to have a finite, positive, pure delay, as is shown in figure 1.9. Proper operation of such an asynchronous circuit requires the following conditions.

• Because of delays, a combinational circuit may produce a transient error or spike, called a *hazard.* Such an error, if applied to the input of an unclocked flip-flop or latch, may result in a permanent incorrect state. Hence the combinational logic portion of the circuit should be designed to be hazard-free.

• The inputs are constrained so as to change only when the memory elements are all in *stable* conditions ($y_i = Y_i$ for all i). This is called the *fundamental mode operation.*

• A situation whereby more than one state variable must change in the course of a transition is called a *race* condition. If correct behavior of the circuit depends upon the outcome of the race, then it is called a *critical race.* To ensure that the operation of the circuit is not affected by transients, the critical race should be avoided by making the proper state assignment.

1.2 Fault Modeling

Logic gates are realized by transistors, which are classified into bipolar transistors and metal oxide semiconductor field-effect transistors (MOSFET, or simply MOS). The logic families based on bipolar transistors are transistor-transistor logic (TTL), emitter-coupled logic (ECL), and so forth. Some logic families based on MOSFET are p-channel MOSFET (p-MOS), n-channel MOSFET (n-MOS), and complementary MOSFET (CMOS). Although ECL and TTL are important for high-speed applications, their integration sizes are limited by the heat generated by their heavy power consumption and by large gate sizes. In contrast, the MOS logic families are well suited for LSI or VLSI, because larger integrations can be obtained with them than with bipolar logic families. Most LSI and VLSI circuits of today are implemented by MOS. However, the increasing use of MOS technology for LSI and VLSI circuits has introduced new testing problems, as will be seen later in this chapter.

A *fault* of a circuit is a physical defect of one or more components. Faults can be classified as logical or parametric. A *logical fault* is one that causes the logic function of a circuit element or an input signal to be changed to some other logic function; a *parametric fault* alters the magnitude of a circuit parameter, causing a change in some factor such as circuit speed, current, or voltage.

Circuit malfunctions associated with timing are due mainly to circuit

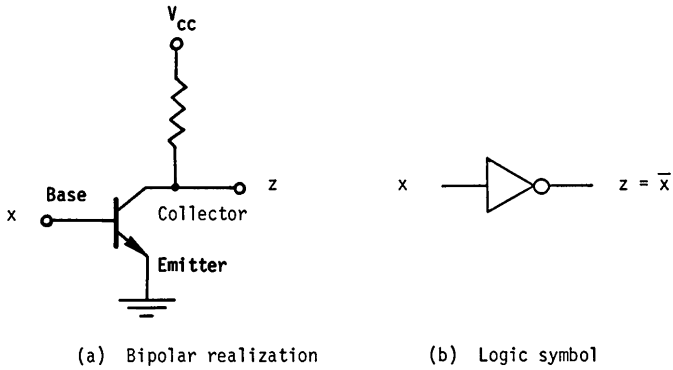


Figure 1.10
Inverter

delays. Those faults that relate to circuit delays such as slow gates are called *delay faults.* Usually, delay faults only affect the timing operation of the circuit, which may cause hazards or critical races.

Faults that are present in some intervals of time and absent in others are *intermittent faults.* Faults that are always present and do not occur, disappear, or change their nature during testing are called *permanent faults* or *solid faults.* Although many intermittent faults eventually become solid, the early detection of intermittent faults is very important to the reliable operation of the circuit. However, there are no reliable means of detecting their occurrence, since such a fault may disappear when a test is applied. In this book, we will consider mainly logical and solid faults.

Figure 1.10 shows an n-p-n transistor implementing an inverter. When the input x is a high voltage, the output z is a low voltage; when x is a low voltage, z is a high voltage. An open collector or base of the transistor would cause the output z to be permanently high, i.e., stuck at 1 (s-a-1). On the other hand, a short circuit between the collector and the emitter would cause z to be permanently low, i.e., stuck at 0 (s-a-0). These faults are called *stuck-at faults.*

Faults in which two lines are shorted are called *bridging faults.* The technology used will determine what effect the bridging will have. Generally, either high or low will dominate. If two output lines are shorted and low dominates, both are replaced by the AND gate of the two lines, as shown in figure 1.11. This effect is the same as the Wired-AND usually used in TTL gates. (Figure 1.12 shows the Wired-AND logic used in TTL gates.)

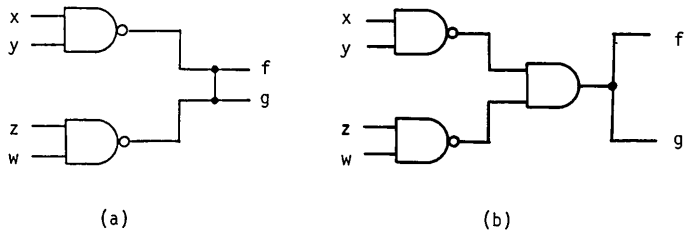


Figure 1.11
AND-type bridging fault

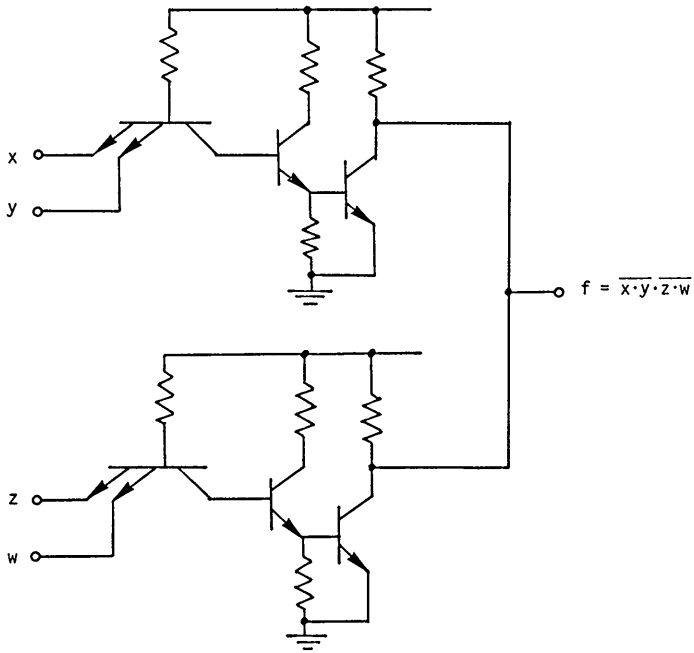


Figure 1.12
Wired-AND used in TTL gates

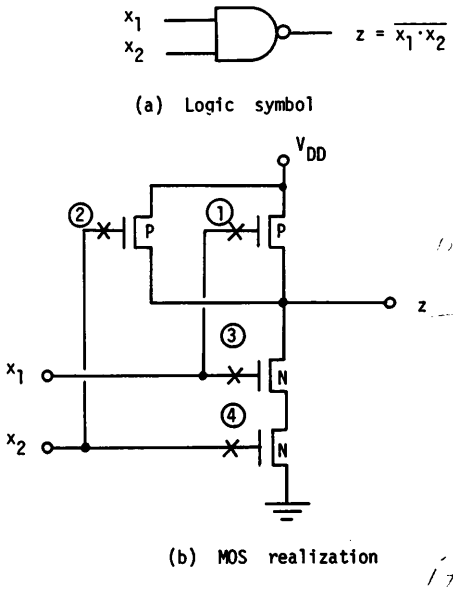


Figure 1.13
CMOS two-input NAND gate

If high dominates, both of the lines are replaced by the OR gate of the two lines. The ECL gate has the feature that the OR of the outputs can be realized simply by tying together these outputs. Hence, in logic circuits implemented by ECL gates, bridging faults cause the affected signals to be ORed.

For most practical purposes, logical faults are successfully modeled by stuck-at faults or bridging faults. However, not all faults can be modeled by these classical faults. This is illustrated by the following examples. Figure 1.13 shows a CMOS two-input NAND gate with p-MOS and n-MOS FETs. The output z is a low voltage if and only if both inputs x_1 and x_2 are high. In figure 1.13, four possible open faults (numbered 1 through 4) are indicated. The first fault, numbered 1, is caused by an open, or missing, p-channel x_1-input pull-up transistor. Under this fault, when the input x_1 is low and the input x_2 is high, the output z becomes an undesired, high-impedance state and retains the logic value of the previous output state. The length of time the state is retained, however, is determined by the leakage current at the node. (Table 1.1 is the truth table for the two-input CMOS NAND gate for both the fault-free condition and the three faulted

Table 1.1
Truth table for CMOS NAND gate

x_1	x_2	z normal	z open at ①	z open at ②	z open at ③ or ④
0	0	1	1	1	1
0	1	1	Previous state	1	1
1	0	1	1	Previous state	1
1	1	0	0	0	Previous state

conditions.) Consequently, these open faults cause combinational circuits to become *sequential*, and thus they cannot be modeled as classical (e.g., stuck-at) faults.

Crosspoint faults in programmable logic arrays (PLAs) also cannot be modeled by the classical faults. Figure 1.14(a) shows an implementation of a PLA in MOS technology. PLAs usually realize two-level AND-OR logic circuits. Figure 1.14(b) shows the two-level AND-OR logic circuit equivalent to the PLA of figure 1.14(a). A PLA inherently has a device (a diode or a transistor) at every crosspoint in the arrays, even if it may not be used. The connection of each device is programmed to realize the desired logic. A crosspoint fault can be caused in a PLA by an extra or a missing device. Although most of the crosspoint faults can be modeled by stuck-at faults, there still exist some crosspoint faults that cannot be modeled by stuck-at faults.

Consider an extra device at the crosspoint A shown in figure 1.14(a). In the absence of a fault, the output realizes the logic function

$$z = x_1x_2 + \bar{x}_1\bar{x}_2x_3.$$

If a transistor appears at A, the first product term x_1x_2 will shrink to $x_1x_2x_3$. Hence, the faulted function will be

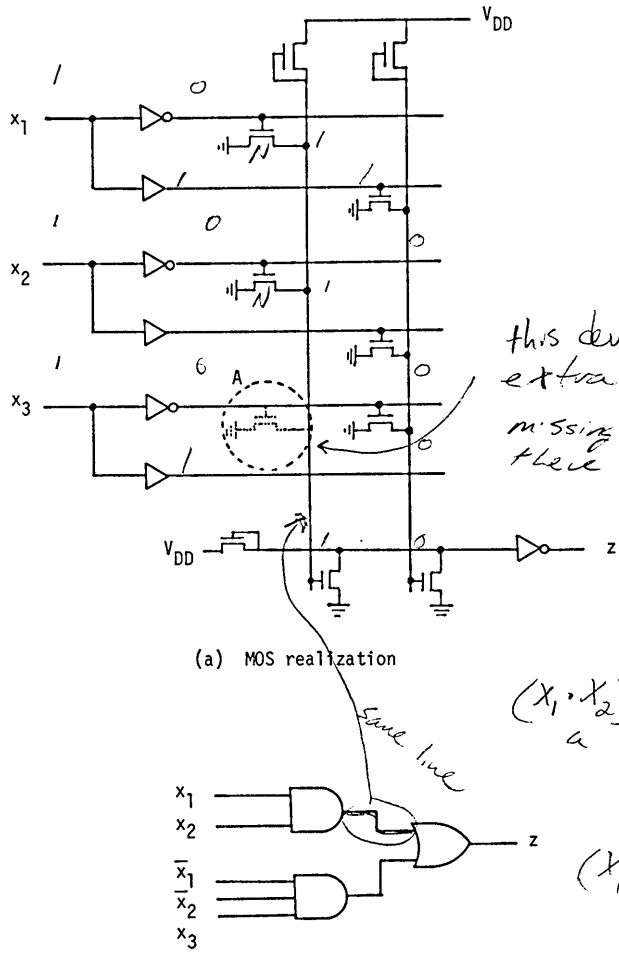
$$z = x_1x_2x_3 + \bar{x}_1\bar{x}_2x_3.$$

This function cannot be caused by any stuck-at fault in the equivalent AND-OR logic circuit of figure 1.14(b).

1.3 Testing Problems

To ensure the proper operation of a system, we must be able to detect a fault when one has occurred and to locate it or isolate it to a specific

CROSS Point faults



this device is known as extra device

missing if supposed to be there

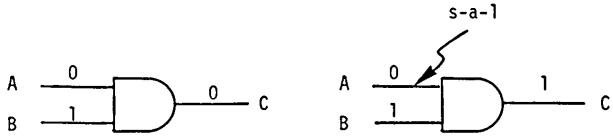
$(x_1 \cdot x_2) + (\bar{x}_1 \cdot \bar{x}_2 \cdot x_3) = z$

a b

with fault

$(x_1 \cdot x_2 \cdot x_3) + (\bar{x}_1 \cdot \bar{x}_2 \cdot x_3) = z$

Figure 1.14
Programmable logic array



(a) Fault-free AND gate (b) Faulty AND gate.

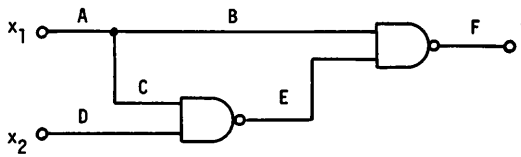
Figure 1.15
Test for stuck-at fault

component—preferably an easily replaceable one. The former procedure is called *fault detection*, and the latter is called *fault location*, *fault isolation*, or *fault diagnosis*. These tasks are accomplished with tests. A *test* is a procedure to detect and/or locate faults. Tests are categorized as fault-detection tests or fault diagnostic tests. A fault-detection test tells only whether a circuit is faulty or fault-free; it tells nothing about the identity of a fault if one is present. A fault diagnostic test provides the location and the type of a fault and other information. The quantity of information provided is called the *diagnostic resolution* of the test; a fault-detection test is a fault diagnostic test of zero diagnostic resolution.

Logic circuits are tested by applying a sequence of input patterns that produce erroneous responses when faults are present and then comparing the responses with the correct (expected) ones. Such an input pattern used in testing is called a *test pattern*. In general, a test for a logic circuit consists of many test patterns. They are referred to as a *test set* or a *test sequence*. The latter term, which means a series of test patterns, is used if the test patterns must be applied in a specific order. Test patterns, together with the output responses, are sometimes called *test data*.

Figure 1.15(a) shows a fault-free AND gate. Figure 1.15(b) shows a faulty AND gate in which input A is stuck-at-1. The input pattern applied to the fault-free AND gate has an output value of 0. In contrast, the output value of the faulty AND gate is 1, since the stuck-at-1 fault on A creates the erroneous response. There is a definite difference between the faulty gate and the fault-free gate. Therefore, the pattern 01 shown in figure 1.15 is a test pattern for the A s-a-1 fault.

If there exists only one fault in a circuit, it is called a *single fault*. If there exist two or more faults at the same time, then the set of faults is called a *multiple fault*. For a circuit with k lines, there are at most $2k$ possible single



total single faults = 6K
total mult. faults = 3K-1-2K

Figure 1.16

A/0 + F/1 are equivelant
there are others also

Table 1.2
All single stuck faults

x_1	x_2	z	A/0	B/0	C/0	D/0	E/0	F/0	A/1	B/1	C/1	D/1	E/1	F/1
0	0	1	1	1	1	1	1	0	0	0	1	1	1	1
0	1	1	1	1	1	1	1	0	1	0	1	1	1	1
1	0	0	1	1	0	0	1	0	0	0	0	1	0	1
1	1	1	1	1	0	0	1	0	1	1	1	1	0	1

inputs, output no faults, 1 if fault shown occurs, C/1 cannot be tested

stuck-at faults. For multiple faults, the number of possible faults increases dramatically to $3^k - 1$, since any line may be fault-free, s-a-0, or s-a-1. A circuit with 100 lines would contain approximately 5×10^{47} faults. This would be far too many faults to assume, and hence testing for multiple faults would be impractical. However, for most approaches the model of a single stuck-at fault has proved to provide the best practical basis. For example, every complete single-fault-detection test set in any internal fanout-free combinational circuit is known to cover at least 98 percent of all multiple faults made up of six or fewer faults (Agarwal and Fung 1981).

2K ← single faults

Since several different faults often may cause a circuit to malfunction in precisely the same way, it is convenient to group these *equivalent faults* (or *indistinguishable faults*) into equivalence classes. In the circuit shown in figure 1.16, there are six signal lines (nets); hence, there are at most twelve possible single stuck-at faults. Table 1.2 illustrates a truth table with columns listing all possible single stuck-at faults and rows indicating all input patterns, where a line A stuck-at-0 fault, for example, is denoted as A/0. As can be seen from table 1.2, the malfunctions caused by A/0, B/0, E/0, D/1, and F/1 are all the same, so they are grouped together in a fault equivalence class. Similarly, C/0, D/0, and E/1 are grouped together, as shown in table 1.3. A fault is called *redundant* if its presence causes no malfunction. In other words, the output function of a circuit with a redun-

Table 1.3
Fault equivalence classes

f_0	z (fault-free), C/1
f_1	A/0, B/0, E/0, D/1, F/1
f_2	C/0, D/0, E/1
f_3	F/0
f_4	A/1
f_5	B/1

Table 1.4
Fault table for representative faults

x_1	x_2	f_1	f_2	f_3	f_4	f_5
0	0			x	x	x
0	1			x		x
1	0	x				
1	1		x	x		

dant fault is exactly the same as that of the fault-free circuit. In the circuit of figure 1.16, fault C/1 is redundant, as table 1.3 shows.

In fault detection and location, only the faults that are picked out as *representative faults* from each fault equivalence class need be considered, since any two equivalent faults are indistinguishable from input-output behavior only. Table 1.4, which is a *fault table* for five representative faults of table 1.3, indicates precisely which test pattern will detect each fault. Finding a fault-detection test set is simply a matter of defining a set of rows, each indicating a test pattern, so that each column indicating a fault class may have an "x" in one of the rows. For this example we obtain {00, 10, 11} as a fault-detection test set. A fault diagnostic test set is obtained by defining a set of rows so that no pair of column patterns corresponding to the rows may be the same. From table 1.4 we have {00, 01, 10, 11} as the diagnostic test set.

The testing of logic circuits is performed in two main stages: generating test patterns for a circuit under test (the *test generation* stage) and applying the test patterns to the circuit (the *test application* stage). Thus, the generation of test patterns is important; however, it is very difficult for large (e.g.

LSI and VLSI) circuits, so most of the effort of the past 20 years in this field went into research and development on efficient and economical test-generation procedures.

The quality of a test (a set or a sequence of test patterns) depends much on the fault coverage as well as the size or length of the test. The *fault coverage* (or *test coverage*) of a test is the fraction of faults that can be detected or located within the circuit under test. The fault coverage of a given test is determined by a process called *fault simulation*, in which every given test patterns is applied to a fault-free circuit and to each of the given faulty circuits, each circuit behavior is simulated, and each circuit response is analyzed to find what faults are detected by the test pattern. Fault simulation is also used to produce *fault dictionaries*, in which the information needed to identify a faulty element or component is gathered.

Summary

The test-generation process includes fault modeling and reduction, test-pattern generation, fault simulation, fault-coverage evaluation, and the production of a fault dictionary. The first step consists of developing a fault dictionary for the circuit (that is, modeling the faults that are assumed) and reducing the number of faults in terms of the fault equivalence relation. Usually, the "single stuck-at fault" model is adopted, and the fault dictionary is generated directly from the logic-circuit description by arranging the distinguishable faults for each gate in tabular form. Next, test patterns are generated to test for the set of faults listed in the fault dictionary. The test patterns are then simulated against the faulted circuits in the fault dictionary, and the fault coverage is evaluated from the results of the fault simulation (which include lists of tested and untested faults). If the fault coverage is inadequate, then the process of test-pattern generation and fault simulation is repeated for the untested faults until an adequate fault coverage is achieved. Finally, the fault dictionary is completed by specifying enough information to detect and locate the faults.

To be practical and cost-effective, the above test-generation process is generally automated in the form of a collection of application software. In particular, the test-pattern generation and the fault simulation should interact effectively to give a high level of fault coverage at a low computation cost. The application of LSI and VLSI requires more efficient *automatic test generation* (ATG) systems.

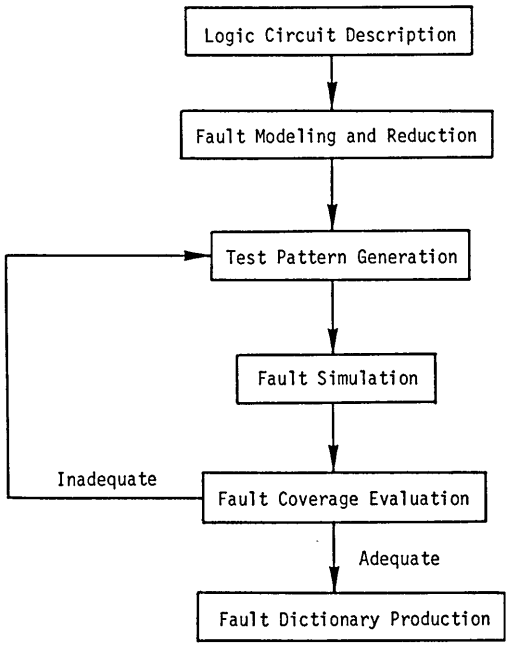


Figure 1.17
Procedure for test generation

1.4 Testing Schemes

Logic testing is performed in various stages, including factory testing of chips, boards, and systems and field testing of boards and systems during periodic maintenance and repair. Usually, fault detection comes first. If it is determined that a fault is present, fault diagnosis is then used to isolate the faulty node or component for repair. In the testing of LSI and VLSI chips it is not necessary to locate faulty gates, since the entire chip must be discarded if any output value is faulty. However, fault diagnosis is required in the testing of boards and systems, and diagnosis at the system level is very important in factory testing and in field maintenance. After a fault has been isolated at the system level, components or individual modules are replaced until a repair is accomplished.

Testing approaches are distinguished by the techniques used to generate and process test data, and can be divided into two categories: on-line and

off-line. *On-line testing* is executed during system run time, concurrent with normal computer operation. Data patterns from normal computation serve as test patterns, and a failure caused by some fault can be detected by built-in monitoring circuits. Since testing and normal computation can proceed concurrently, on-line testing is also called *concurrent* testing and is effective against intermittent faults. *Off-line* testing is executed when the system under test is off line. In off-line testing, specific test patterns are usually provided; data patterns from normal computation cannot serve as test patterns.

Testing approaches are also divided into external testing and built-in testing. In *external testing* the test equipment is external to the system under test; that is, test patterns are applied by the external tester and the responses are then evaluated. In *built-in testing* the test equipment is built into the system under test. As described below, on-line testing uses built-in monitoring circuits to detect a failure caused by some intermittent or solid fault, and thus it belongs to the category of built-in testing. Off-line testing encompasses both types of testing schemes, external and built-in.

On-line testing schemes are usually implemented by *redundancy techniques*: information redundancy and hardware redundancy. Information-redundancy approaches include such popular coding schemes as parity, cyclic redundancy checks, and error-correcting codes; hardware-redundancy techniques include self-checking circuits at the gate level, duplication at the module level, and replicated computers at the system level.

The most widely used linear codes are odd and even parity check codes for single-bit fault detection and Hamming codes for multiple-bit fault detection. In those error-detecting codes, one or more redundant bits are simply appended to detect errors. For example, consider a combinational circuit with n inputs and m outputs. If only $k < 2^m$ output values can occur during normal operation, the occurrence of any of $2^m - k$ unallowable values indicates a malfunction caused by a fault in the circuit. The output values that do occur are called *code words*, and unallowable output values are called *noncode words*. Error-detecting codes are usually used to design hardware that can automatically detect errors in the circuit (figure 1.18).

Self-checking circuits are those circuits in which the occurrence of a fault can be recognized only by observing the outputs of the circuit. An important subclass of self-checking circuits is *totally self-checking circuits*, defined as follows. A circuit is *fault-secure* for a set of faults F if, for any fault in F and any allowable code input, the output is a noncode word or the correct

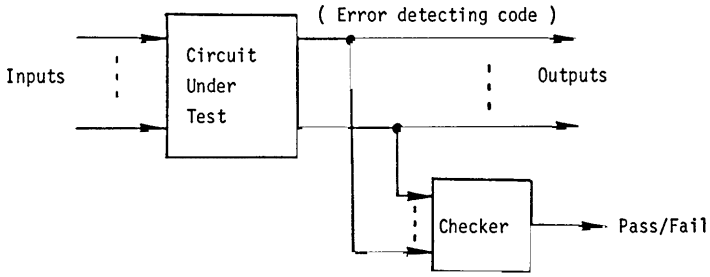


Figure 1.18
On-line testing using information redundancy

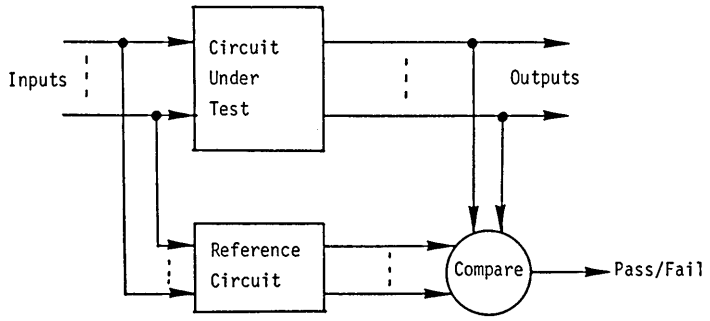


Figure 1.19
On-line testing using hardware redundancy

code word, never an incorrect code word. In other words, the output of the faulty circuit cannot be a code word and, at the same time, be different from the correct output. Thus, as long as the output is a code word, it can safely be assumed to be correct. On the other hand, a circuit is *self-testing* for a set of faults F if, for any fault f in F, there exists an allowable code input that detects f (i.e., the resulting output is a noncode word). In other words, the input set contains at least one test for every fault in the prescribed set. A totally self-checking circuit is both fault-secure and self-testing for all faults under consideration. Hence, a totally self-checking circuit can guarantee that any fault in the circuit cannot cause an error in the output without detection of the fault.

One of the most effective approaches to on-line testing using hardware redundancy is *dual redundancy*; that is, straightforward duplication of the resource being tested. Figure 1.19 shows this duplication scheme for built-in

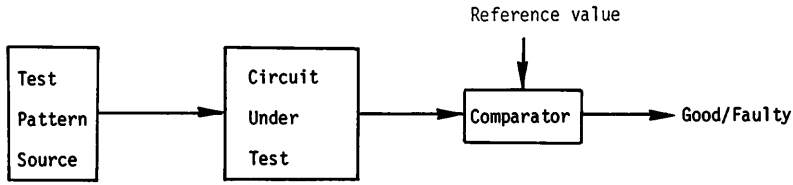


Figure 1.20
General testing scheme

testing. Although full duplication is the upper bound on hardware redundancy for fault detection and thus requires more hardware than other hardware-redundancy approaches, the duplication scheme can easily be adopted at any part of the computer system and at any level within the computer hierarchy.

In the external-testing approaches, the circuit under test (CUT) is tested with *automatic test equipment* (ATE) in which a sequence of test patterns are applied to the CUT and the responses of the CUT are compared against reference values (correct responses). Figure 1.20 shows the general scheme of testing. Test patterns and reference values are produced by either software-based or hardware-based methods. In the software-based scheme, test patterns and correct responses are generated in advance and stored in memory. Test patterns are produced either manually, by a design or test engineer, or automatically, by test-generation programs. In contrast, both test patterns and reference values can be produced each time a CUT is tested by hardware-implemented algorithms. In these hardware-based schemes, test patterns are usually generated randomly or pseudorandomly and simultaneously applied to the CUT and a known good circuit, called a *gold circuit*. The output responses of the CUT are then compared with those of the gold circuit. This type of testing method (figure 1.21) is called *random testing*, and in contrast the former software-based testing is called *deterministic testing*. Deterministic testing requires costly or time-consuming test generation and memory to store huge amounts of test data, where as random testing eliminates the cost and the time requirement of generating or storing test data. However, random testing has disadvantages: The need for a gold circuit may be bothersome, and the reliability of the gold circuit is not guaranteed. Synchronization of the two circuits and fault coverage of random test patterns may also cause problems.

In the above-mentioned testing schemes, the total responses are com-

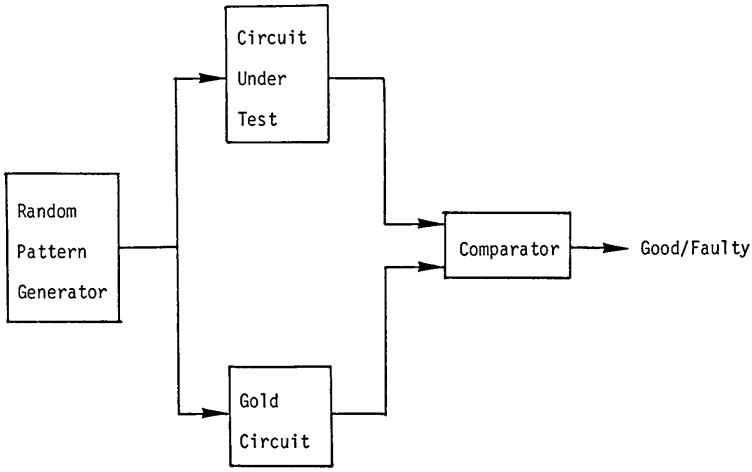


Figure 1.21
Random testing

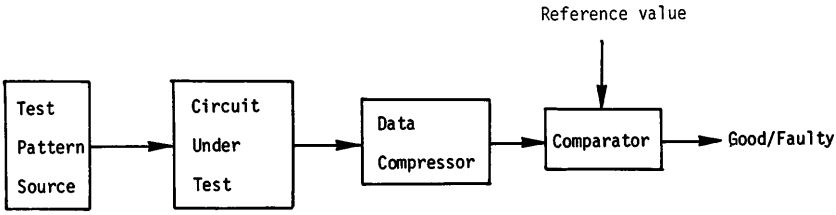


Figure 1.22
Compact testing

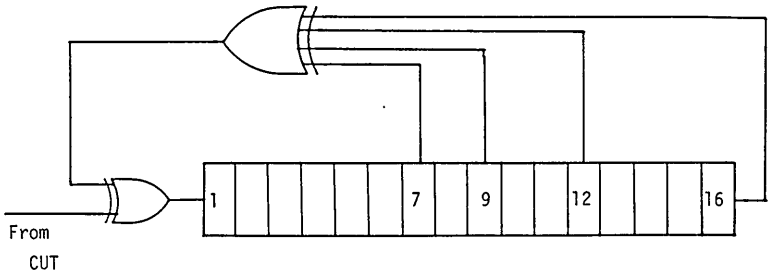


Figure 1.23
A 16-bit linear feedback shift register (Frohwerk 1977). © Copyright 1977 Hewlett-Packard Company. Reproduced with permission.

pared with the correct reference values, which are usually high volumes. This difficulty of analysis and storage of huge amounts of response data can be avoided through an approach called *compact testing*, in which, rather than the total responses, compressed response data are used for comparison. Figure 1.22 shows the general scheme for compact testing. The model of figure 1.22 allows wide variations in the methods for test-pattern generation and data compression. In a global sense, compact testing methods may be classified as either deterministic or random in connection with the techniques used to generate test patterns. The data compressor can be implemented with simple circuitry, such as counters and linear-feedback shift registers. Since compact testing requires little test equipment, it is suited for built-in testing. A widely used method of compact testing is *signature analysis*, which compresses the output response through a 16-bit linear feedback shift register whose contents are called the *signature*. Figure 1.23 shows a 16-bit linear-feedback shift register.