8. FUNCTIONAL TESTING

About This Chapter

The test generation methods studied in the previous chapters are based on a structural model of a system under test and their objective is to produce tests for structural faults such as stuck faults or bridging faults. But detailed structural models of complex devices are usually not provided by their manufacturers. And even if such models were available, the structure-based test generation methods are not able to cope with the complexity of VLSI devices. In this chapter we examine *functional testing methods* that are *based on a functional model* of the system. (Functional test generation methods for SSFs using RTL models for components are described in Chapter 6.) We discuss only external testing methods.

8.1 Basic Issues

A functional model reflects the functional specifications of the system and, to a great extent, is independent of its implementation. Therefore functional tests derived from a functional model can be used not only to check whether physical faults are present in the manufactured system, but also as design verification tests for checking that the implementation is free of design errors. Note that tests derived from a structural model reflecting the implementation cannot ascertain whether the desired operation has been correctly implemented.

The objective of functional testing is to *validate the correct operation of a system with respect to its functional specifications.* This can be approached in two different ways. One approach assumes specific functional fault models and tries to generate tests that detect the faults defined by these models. By contrast, the other approach is not concerned with the possible types of faulty behavior and tries to derive tests based only on the specified fault-free behavior. Between these two there is a third approach that defines an implicit fault model which assumes that almost any fault can occur. Functional tests detecting almost any fault are said to be *exhaustive*, as they must completely exercise the fault-free behavior. Because of the length of the resulting tests, exhaustive testing can be applied in practice only to small circuits. By using some knowledge about the structure of the circuit and by slightly narrowing the universe of faults guaranteed to be detected, we can obtain *pseudoexhaustive tests* that can be significantly shorter than the exhaustive ones. The following sections discuss these three approaches to functional testing: (1) functional testing without fault models, (2) exhaustive and pseudoexhaustive testing, and, (3) functional testing using specific fault models.

8.2 Functional Testing Without Fault Models

8.2.1 Heuristic Methods

Heuristic, or ad hoc, functional testing methods simply attempt to exercise the functions of the system. For example, a functional test for a flip-flop may include the following:

1. Validate that the flip-flop can be set (0 to 1 transition) and reset (1 to 0 transition).
2. Validate that the flip-flop can hold its state.

The following example [Chiang and McCaskill 1976] illustrates a heuristic testing procedure for a microprocessor.

Example 8.1: The operation of a microprocessor is usually defined by its architectural block diagram together with its instruction set. Figure 8.1 shows the block diagram of the INTEL 8080. We assume that an external tester is connected to the data, address, and control busses of the 8080. The tester supplies the 8080 with instructions to be executed and checks the results.

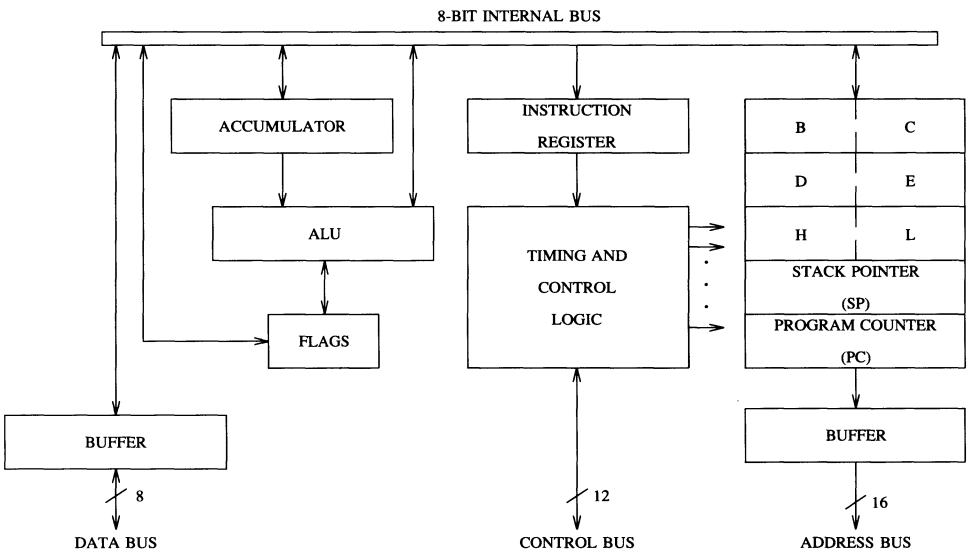


Figure 8.1 Architecture of the INTEL 8080 microprocessor

A typical functional test consists of the following steps:

1. test of the program counter (PC):
   a. The tester resets the 8080; this also clears the PC.
   b. The tester places a NOP (no operation) instruction on the data bus and causes the 8080 to execute it repeatedly. The repeated execution of NOP causes the PC to be incremented through all its 2^{16} states. The content of the PC is available for checking on the address bus.
2. test of registers H and L:
   a. The tester writes 8-bit patterns into H and L using MVI (move-immediate) instructions.

b. Executing an PCHL instruction transfers the contents of H and L into the PC. Here we rely on having checked the PC in step 1.

This sequence is repeated for all the possible 256 8-bit patterns.

3. test of registers B,C,D, and E:

   In a similar fashion, the tester writes an 8-bit pattern into a register R$\in${B,C,D,E}. Then R is transferred into the PC via H or L (R cannot be directly transferred into the PC). Here we take advantage of the testing of the PC and HL done in steps 1 and 2. These tests are executed for all the 256 8-bit patterns.

4. test of the stack pointer (SP):

   The SP is incremented and decremented through all its states and accessed via the PC.

5. test of the accumulator:

   All possible patterns are written into the accumulator and read out. These can be done directly or via previously tested registers.

6. test of the ALU and flags:

   This test exercises all arithmetic and logical instructions. The operands are supplied directly or via already tested registers. The results are similarly accessed. The flags are checked by conditional jump instructions whose effect (the address of the next instruction) is observed via the PC.

7. test of all previously untested instructions and control lines. □

An important issue in the functional testing of a microprocessor is whether its instruction set is *orthogonal*. An orthogonal instruction set allows every operation that can be used in different addressing modes to be executed in every possible addressing mode. This feature implies that the mechanisms of op-code decoding and of address computation are independent. If the instruction set is not orthogonal, then every operation must be tested for all its addressing modes. This testing is not necessary for an orthogonal instruction set, and then the test sequences can be significantly shorter.

The Start-Small Approach

Example 8.1 illustrates the *start-small* (or *bootstrap*) approach to functional testing of complex systems, in which the testing done at a certain step uses components and/or instructions tested in previous steps. In this way the tested part of the system is gradually extended. The applicability of the start-small approach depends on the degree of overlap among the components affected by different instructions. Its objective is to simplify the fault location process. Ideally, if the first error occurs at step i in the test sequence, this should indicate a fault in the new component(s) and/or operation(s) tested in step i.

A technique for ordering the instructions of a microprocessor according to the start-small principle is presented in [Annaratone and Sami 1982]. The *cardinality* of an instruction is defined as the number of registers accessed during the execute phase of an instruction (i.e., after the fetch-and-decode phase). Thus NOP, which has only

the fetch-and-decode phase, has cardinality 0. Instructions are also graded according to their *observability*; the observability of an instruction shows the extent to which the results of the register operations performed by the instruction are directly observable at the primary outputs of the microprocessor. Instructions are tested in increasing order of their cardinality. In this way the instructions affecting fewer registers are tested first (a classical "greedy" algorithm approach). Among instructions of the same cardinality, priority is given to those with higher observability.

The Coverage Problem

The major problem with heuristic functional testing is that the quality of the obtained functional tests is unknown. Indeed, without a fault model it is extremely difficult to develop a rigorous quality measure.

Consequently an important question is whether a heuristically derived functional test does a good job of detecting physical faults. For cases in which a low-level structural model reflecting the implementation of the system is available, experience shows that the fault coverage of a typical heuristic functional test is likely to be in the 50 to 70 percent range. Hence, in general, such a test does not achieve a satisfactory fault coverage, but it may provide a good basis that can be enhanced to obtain a higher-quality test. However, low-level structural models are usually not available for complex systems built with off-the-shelf components.

Heuristic measures can be used to estimate the "completeness" of a test with respect to the control flow of the system. These measures are based on monitoring the activation of operations in an RTL model [Noon 1977, Monachino 1982]. For example, if the model contains a statement such as

$$\textbf{if } x \textbf{ then } operation_1 \textbf{ else } operation_2$$

the technique determines if the applied test cases make the condition x both true and false during simulation. A "complete" test is required to exercise all possible exits from decision blocks. One measure is the ratio between the number of exits taken during simulation and the number of all possible exits. A second, more complicated measure traces decision paths, i.e., combinations of consecutive decisions. For example, if the above statement is followed by

$$\textbf{if } y \textbf{ then } operation_3 \textbf{ else } operation_4$$

then there are four possible decision paths of length 2 (see Figure 8.2), corresponding to the four combinations of values of x and y. The second measure is the ratio between the number of decision paths of length k traversed during simulation and the total number of such paths. Note that data operations are not addressed by these measures.

An important aspect of functional testing, often overlooked by heuristic methods, is that in addition to verifying the specified operations of a system, it is also necessary to check that unintended operations do not occur. For example, in addition to a correct transfer of data into register R1, the presence of a fault may cause the same data to be written into register R2. In such a case, checking only the desired operation — as is usually done by heuristic methods — is clearly not enough.

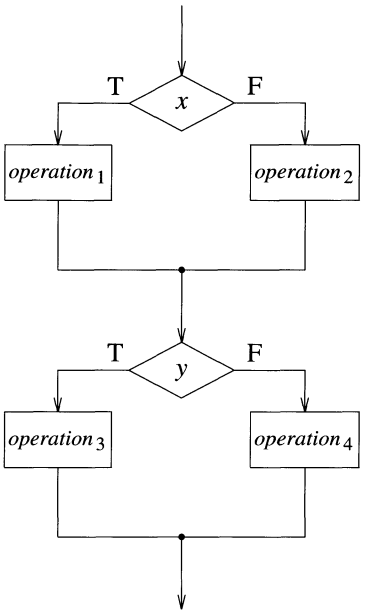


Figure 8.2 Illustration for decision paths tracing

8.2.2 Functional Testing with Binary Decision Diagrams

In Chapter 2 we have introduced *binary decision diagrams* as a functional modeling tool. In this section we describe a functional testing technique based on binary decision diagrams [Akers 1978].

First we recall the procedure used to determine the value of a function f, given its binary decision diagram and values of its inputs. We enter the diagram on the branch labeled f. At an internal node whose label is i we take the left or the right branch depending on whether the value of the variable i is 0 or 1. The *exit value* of the path followed during the traversal is the value, or the value of the variable, encountered at the end of the path. The *inversion parity* of the path is the number of inverting dots encountered along the path, taken modulo 2. For a traversal along a path with exit value v and inversion parity p, the value of the function f is $v \oplus p$. For example, consider the *JK* F/F and its diagram shown in Figure 8.3 (q represents the state of the F/F). The value of y along the path determined by S=0, R=0, C=1, and q=1 is $K \oplus 1 = \overline{K}$.

A traversal of a binary decision diagram implies a certain setting of the variables encountered along the path. Such a traversal is said to define a *mode of operation* of the device. A path whose exit value is x denotes an illegal mode of operation, in which the output value cannot be predicted. For the *JK* F/F of Figure 8.3, setting S=1 and R=1 is illegal. Figure 8.4 lists the five legal modes of operation of the *JK* F/F corresponding to the five paths with non-x exit values.

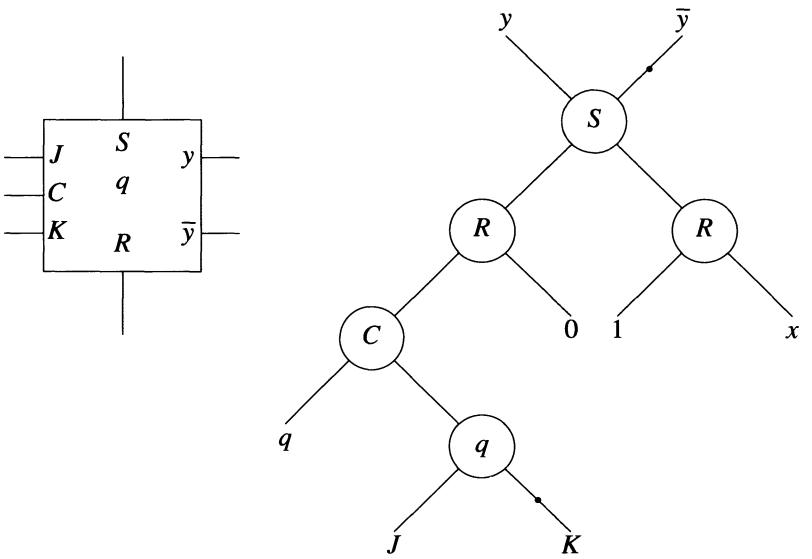


Figure 8.3 Binary decision diagram for a *JK* F/F

S	R	C	q	J	K	y
0	1	x	x	x	x	0
1	0	x	x	x	x	1
0	0	0	q	x	x	q
0	0	1	0	J	x	J
0	0	1	1	x	K	$\overline{K}$

Figure 8.4 Testing experiments for a *JK* F/F

Every (legal) mode of operation can be viewed as defining a testing *experiment* that partitions the variables of a function into three disjoint sets:

- *fixed variables*, whose binary values determine the path associated with the mode of operation;
- *sensitive variables*, which directly determine the output value;
- *unspecified variables*, which do not affect the output (their values are denoted by x).

An experiment provides a *partial specification* of the function corresponding to a particular mode of operation. In Figure 8.3, y is a function $y(S,R,C,q,J,K)$. The partial specification of y given by the third experiment in Figure 8.4 is $y(0,0,0,q,x,x)=q$.

One can show that the set of experiments derived by traversing all the paths corresponding to an output function provides a *complete specification* of the function. That is, every possible combination of variables is covered by one (and only one) experiment. In addition, the experiments are *disjoint*; i.e., every pair of experiments differ in the value of at least one fixed variable.

Some binary decision diagrams may contain adjacent nodes, such as A and B in Figure 8.5, where both branches from A reconverge at B. Consider two traversals involving node A, one with A=0 and the other with A=1. Clearly the results of these two traversals are complementary. Then we can combine the two traversals into one and treat A as a sensitive variable of the resulting experiment. Let v be the result of the traversal with A=0. The result of the combined traversal is $v \oplus A$.

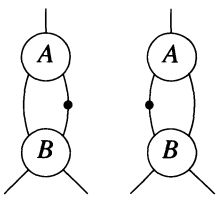


Figure 8.5

Example 8.2: For the diagram of Figure 8.6, let us derive an experiment by following the path determined by A=0, D=1, and C=1. The exit variable of the path is G, and its other sensitive variables are B and F. The result of the traversal with B=0 and F=0 is $G \oplus 1 = \overline{G}$. The result of the combined traversal is $\overline{G} \oplus B \oplus F$. The partial specification of the function $f(A,B,C,D,E,F,G)$ provided by this experiment is $f(0,B,1,1,x,F,G) = \overline{G} \oplus B \oplus F$. □

It is possible that different nodes in a binary decision diagram have the same variable. Hence when we traverse such a diagram we may encounter the same variable more than once. For example, in the diagram of Figure 8.6, when following the path determined by A=1, C=0, and D=1, we reach again a node whose variable is C. Because C has already been set to 0, here we must continue on the left branch. Otherwise, if we continue on the right branch, the resulting experiment would not be *feasible*, because it would specify contradictory values for C. Therefore the traversal process should ensure that only feasible experiments are generated.

An experiment derived by traversing a binary decision diagram is not in itself a test, but only a partial functional specification for a certain mode of operation. Experiments can be used in different ways to generate tests. Since the output value will change with any change of a sensitive variable of an experiment, a useful procedure is to generate all the combinations of sensitive variables for every mode of operation. If the sensitive variables are inputs of the function, this strategy tends to create input-output paths along which many internal faults are also likely to be detected.

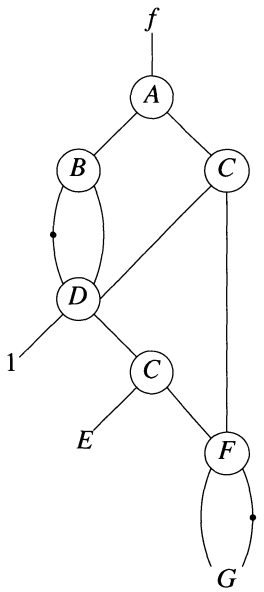


Figure 8.6

Some variables of a function may not appear as sensitive in a set of experiments. This is the case for S and R in Figure 8.4. Now we will illustrate a procedure that generates a test in which an output is made sensitive to a variable s. The principle is to combine two experiments, e_1 and e_2, in which s is the only fixed variable with opposite values, and the output values are (or can be made) complementary. In other words, we need two paths that diverge at a node whose variable is s and lead to complementary exit values. The combined experiment has all the fixed values of e_1 and e_2 (except s). Assume that, using the diagram of Figure 8.3, we want to make y sensitive to the value of S. When S=1, the only legal experiment requires R=0 and produces y=1. To make y sensitive to S, we look for a path with S=0 leading to y=0. (Note that we cannot set R=1, since R has been set to 0.) One such path requires C=0 and q=0. This shows that to make the output sensitive to S we should first reset the F/F.

As an example of functional testing using binary decision diagrams, Akers [1978] has derived the complete set of experiments for a commercial 4-bit ALU with 14 PIs and 8 POs.

Testing of circuits composed of modules described by binary decision diagrams is discussed in [Abadir and Reghbati 1984]. The examples include a commercial 1-bit microprocessor slice. Chang *et al.* [1986] assume a functional fault model in which a fault can alter the path defined by an experiment. Their procedure generates tests with high fault coverage for SSFs. They also show that the test set can be reduced by taking into account a limited amount of structural information.

The main advantage of a binary decision diagram is that it provides a complete and succinct functional model of a device, from which a complete set of experiments — corresponding to every mode of operation of the device — can be easily derived.

8.3 Exhaustive and Pseudoexhaustive Testing

The Universal Fault Model

Exhaustive tests detect all the faults defined by the *universal fault model.* This implicit fault model assumes that any (permanent) fault is possible, except those that increase the number of states in a circuit. For a combinational circuit N realizing the function $Z(x)$, the universal fault model accounts for any fault f that changes the function to $Z_f(x)$. The only faults not included in this model are those that transform N into a sequential circuit; bridging faults introducing feedback and stuck-open faults in CMOS circuits belong to this category. For a sequential circuit, the universal fault model accounts for any fault that changes the state table without creating new states. Of course, the fault universe defined by this model is not enumerable in practice.

8.3.1 Combinational Circuits

To test all the faults defined by the universal fault model in a combinational circuit with n PIs, we need to apply all 2^n possible input vectors. The exponential growth of the required number of vectors limits the practical applicability of this *exhaustive testing* method only to circuits with less than 20 PIs. In this section we present *pseudoexhaustive testing* methods that can test almost all the faults defined by the universal fault model with significantly less than 2^n vectors.

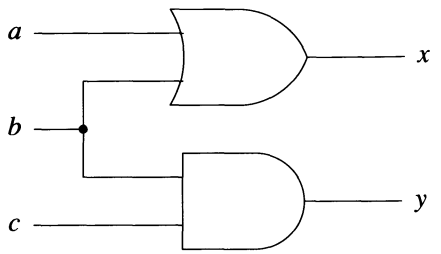
8.3.1.1 Partial-Dependence Circuits

Let $O_1, O_2, \ldots, O_m$ be the POs of a circuit with n PIs, and let n_i be the number of PIs feeding O_i. A circuit in which no PO depends on all the PIs (i.e., $n_i<n$ for all i), is said to be a *partial-dependence circuit.* For such a circuit, pseudoexhaustive testing consists in applying all 2^{n_i} combinations to the n_i inputs feeding every PO O_i [McCluskey 1984].

Example 8.3: The circuit of Figure 8.7(a) has three PIs, but every PO depends only on two PIs. While exhaustive testing requires eight vectors, pseudoexhaustive testing can be done with the four vectors shown in Figure 8.7(b). □

Since every PO is exhaustively tested, the only faults that a pseudoexhaustive test *may* miss are those that make a PO dependent on additional PIs. For the circuit of Figure 8.7(a), the bridging fault $(a.c)$ is not detected by the test set shown. With the possible exception of faults of this type, all the other faults defined by the universal fault model are detected.

Because of the large number of test vectors required in practical circuits, pseudoexhaustive testing is not used in the context of stored-pattern testing; its main use is in circuits employing built-in self-test, where the vectors are generated by hardware embedded in the circuit under test. Pseudoexhaustive testing is discussed in detail in Chapter 11.



(a)

a	b	c
0	0	0
0	1	0
1	0	1
1	1	1

(b)

Figure 8.7 (a) Partial dependence circuit (b) Pseudoexhaustive test set

8.3.1.2 Partitioning Techniques

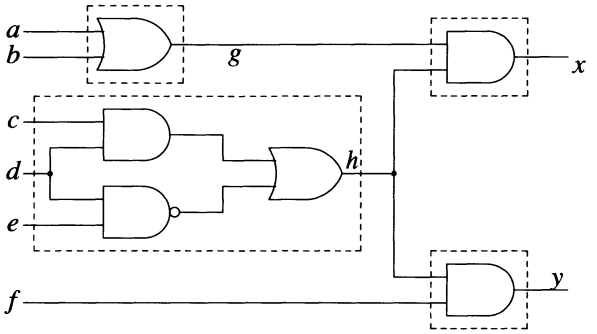
The pseudoexhaustive testing techniques described in the previous section are not applicable to *total-dependence circuits*, in which at least one PO depends on all PIs. Even for a partial-dependence circuit, the size of a pseudoexhaustive test set may still be too large to be acceptable in practice. In such cases, pseudoexhaustive testing can be achieved by *partitioning techniques* [McCluskey and Bozorgui-Nesbat 1981]. The principle is to partition the circuit into *segments* such that the number of inputs of every segment is significantly smaller than the number of PIs of the circuit. Then the segments are exhaustively tested.

The main problem with this technique is that, in general, the inputs of a segment are not PIs and its outputs are not POs. Then we need a means to control the segment inputs from the PIs and to observe its outputs at the POs. One way to achieve this, referred to as *sensitized partitioning*, is based on sensitizing paths from PIs to the segment inputs and from the segment outputs to POs, as illustrated in the following example.

Example 8.4: Consider the circuit in Figure 8.8(a). We partition it into four segments. The first segment consists of the subcircuit whose output is h. The other three segments consist, respectively, of the gates g, x, and y. Figure 8.8(b) shows the eight vectors required to test exhaustively the segment h and to observe h at the PO y. Since h=1 is the condition needed to observe g at the PO x, we can take advantage of the vectors 5 through 8 in which h=1 to test exhaustively the segment g (see Figure 8.8(c)). We also add vectors 9 and 10 to complete the exhaustive test of the segment y. Analyzing the tests applied so far to the segment x, we can observe that the missing combinations are those in which h=0; these can be applied by using vectors 4 and 9.

Figure 8.8(d) shows the resulting test set of 10 vectors. This compares favorably with the 2^6=64 vectors required for exhaustive testing or with the 2^5=32 vectors needed for pseudoexhaustive testing without partitioning. □

In an example presented in [McCluskey and Bozorgui-Nesbat 1981], sensitized partitioning is applied to a commercial 4-bit ALU. This is a total-dependence circuit



(a)

Figure 8.8

with 14 PIs, that would require 2^{14} vectors for exhaustive testing. Its pseudoexhaustive test set based on sensitized partitioning has only 356 vectors.

A pseudoexhaustive test set based on sensitized partitioning detects any fault that changes the truth table of a segment. Since a circuit may have several possible partitions, this fault model depends, to a certain extent, on the chosen set of segments. Partitioning a circuit so that the size of its pseudoexhaustive test set is minimal is an *NP*-complete problem. A partitioning algorithm based on simulated annealing is described in [Shperling and McCluskey 1987].

Note that partitioning techniques assume knowledge of the internal structural model.

8.3.2 Sequential Circuits

For a sequential circuit, the universal fault model accounts for any fault that modifies the state table of the circuit without increasing its number of states. An input sequence that detects every fault defined by this model must distinguish a given n-state sequential machine from all other machines with the same inputs and outputs and at most n states. The existence of such a *checking sequence* is guaranteed by the following theorem [Moore 1956].

Theorem 8.1: For any reduced, strongly connected*, n-state sequential machine M, there exists an input-output sequence pair that can be generated by M, but cannot be generated by any other sequential machine M' with n or fewer states. □

* A reduced machine does not have any equivalent states. In a strongly connected machine any state can be reached from any other state.

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*x*	*y*
1			0	0	0	1		1		1
2			0	0	1	1		1		1
3			0	1	0	1		1		1
4			0	1	1	1		0		0
5			1	0	0	1		1		1
6			1	0	1	1		1		1
7			1	1	0	1		1		1
8			1	1	1	1		1		1

(b)

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*x*	*y*
1			0	0	0	1		1		1
2			0	0	1	1		1		1
3			0	1	0	1		1		1
4			0	1	1	1		0		0
5	0	0	1	0	0	1	0	1	0	1
6	0	1	1	0	1	1	1	1	1	1
7	1	0	1	1	0	1	1	1	1	1
8	1	1	1	1	1	1	1	1	1	1
9			0	1	1	0		0		0
10			0	0	0	0		1		0

(c)

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*x*	*y*
1			0	0	0	1		1		1
2			0	0	1	1		1		1
3			0	1	0	1		1		1
4	0	0	0	1	1	1	0	0	0	0
5	0	0	1	0	0	1	0	1	0	1
6	0	1	1	0	1	1	1	1	1	1
7	1	0	1	1	0	1	1	1	1	1
8	1	1	1	1	1	1	1	1	1	1
9	0	1	0	1	1	0	1	0	0	0
10			0	0	0	0		1		0

(d)

Figure 8.8 (Continued)

Since the generation of a checking sequence is based on the state table of the circuit, this exhaustive testing approach is applicable only to small circuits. We will consider checking sequences only briefly; a comprehensive treatment of this topic can be found in [Friedman and Menon 1971].

A checking sequence for a sequential machine M consists of the following three phases:

1. initialization, i.e., bringing M to a known starting state;
2. verifying that M has n states;
3. verifying every entry in the state table of M.

Initialization is usually done by a *synchronizing sequence*, which, independent of the initial state of M, brings M in a unique state.

The derivation of a checking sequence is greatly simplified if M has a *distinguishing sequence*. Let Z_i be the output sequence generated by M, starting in state q_i, in response to an input sequence X_D. If Z_i is unique for every i=1,2...,n, then X_D is a distinguishing sequence. (A machine M that does not have a distinguishing sequence can be easily modified to have one by adding one output.) The importance of X_D is that by observing the response of M to X_D we can determine the state M was in when X_D was applied.

To verify that M has n distinct states, we need an input/output sequence that contains n subsequences of the form X_D/Z_i for i=1,2,...,n. We can then use X_D to check various entries in the state table. A transition $N(q_i,x)=q_j$, $Z(q_i,x)=z$, is verified by having two input/output subsequences of the following form:

$$X_D X' X_D / Z_p Z' Z_i \quad \text{and} \quad X_D X' x X_D / Z_p Z' z Z_j$$

The first one shows that $X_D X'$ takes M from state q_p to state q_i. Based on this, we can conclude that when input x is applied in the second subsequence, M is in state q_i. The last X_D verifies that x brings M to state q_j.

The sequences that verify that M has n distinct states and check every entry in the state table can often be overlapped to reduce the length of the checking sequence.

8.3.3 Iterative Logic Arrays

Pseudoexhaustive testing techniques based on partitioning are perfectly suited for circuits structured as *iterative logic arrays* (ILAs), composed from identical *cells* interconnected in a regular pattern as shown in Figure 8.9. The partitioning problem is naturally solved by using every cell as a segment that is exhaustively tested. We consider *one-dimensional* and *unilateral* ILAs, that is, ILAs where the connections between cells go in only one direction. First we assume that cells are combinational. A ripple-carry adder is a typical example of such an ILA.

Some ILAs have the useful property that they can be pseudoexhaustively tested with a number of tests that does not depend on the number of cells in the ILA. These ILAs are said to be *C-testable* [Friedman 1973]. The following example shows that the ripple-carry adder is C-testable.

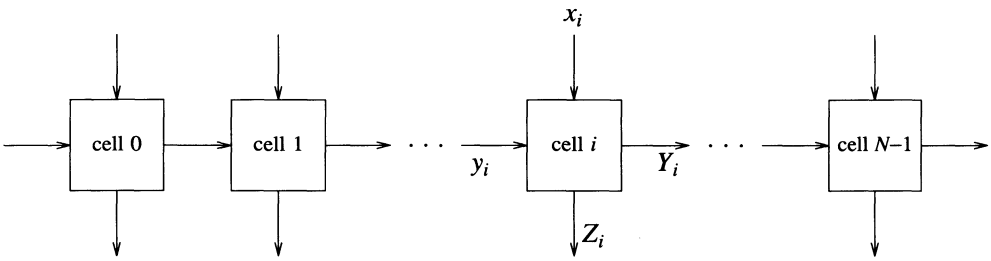
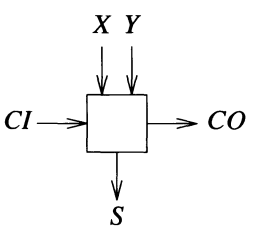


Figure 8.9 Iterative Logic Array

Example 8.5: Consider the truth table of a cell of the ripple-carry adder (Figure 8.10). In this truth table, we separate between the values of the cell inputs that are PIs (X and Y) and those that are provided by the previous cell (CI). An entry in the table gives the values of CO and S. We can observe that CO=CI in six entries. Thus if we apply the X,Y values corresponding to one of these entries to every cell, then every cell will also receive the same CI value. Hence these six tests can be applied concurrently to all cells. In the remaining two entries, CO=$\overline{CI}$. The tests corresponding to these two entries can be applied to alternating cells, as shown in Figure 8.11. Hence a ripple-carry adder of arbitrary size can be pseudoexhaustively tested with only eight tests; therefore, it is a C-testable ILA. □



(a)

	X Y							
CI	0	0	0	1	1	0	1	1
0	0,	0	0,	1	0,	1	1,	0
1	0,	1	1,	0	1,	0	1,	1

CO, S

(b)

Figure 8.10 A cell of a ripple-carry adder

Let us denote an input vector of an ILA with N cells by $(y_0\ x_0\ x_1\ \cdots\ x_{N-1})$, where y_0 is the y input applied to the first cell, and x_i (i=0,1,...,$N-1$) is the x input applied to cell i. Let us denote the corresponding output vector by $(Z_0\ Z_1\ \cdots\ Z_{N-1})$, where Z_i is the response at the primary outputs of cell i. Consider a sequential circuit constructed from one cell of the array by connecting the Y outputs back to the y inputs (Figure 8.12). If this circuit starts in state y_0, its response to the input sequence $(x_0,x_1,...,x_{N-1})$ is the sequence $(Z_0,Z_1,...,Z_{N-1})$. Hence we can use this sequential circuit to represent the ILA. (This is the inverse of the technique described in

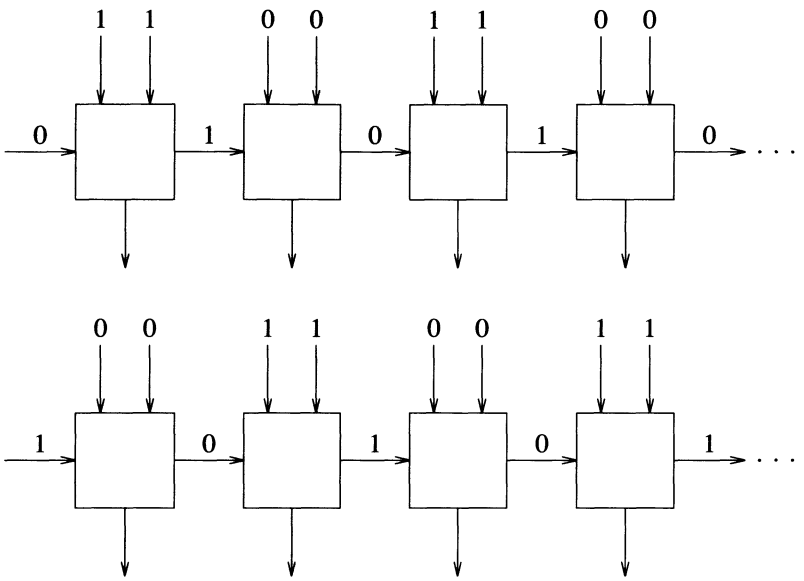


Figure 8.11 Tests with $CO=\overline{CI}$ for a ripple-carry adder

Chapter 6, which uses an ILA to model a sequential circuit.) Verifying the truth table of a cell in the ILA corresponds to verifying every entry in the state table of the equivalent sequential circuit. The truth table of the cell of the ripple-carry adder, shown in Figure 8.10(b), can be interpreted as the state table of the sequential circuit modeling the ripple-carry adder.

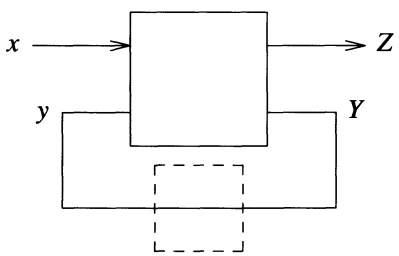


Figure 8.12

Let us consider the checking of an entry (y,x) of the truth table of an ILA cell. To have the total number of tests independent of the number of cells in the ILA, the same input combination (y,x) must be simultaneously applied to cells spaced at regular

intervals — say, k — along the array. (Example 8.5 has k=1 and k=2). For the sequential circuit modeling the ILA, this means that the k-step sequence starting with x, which is applied in state y, must bring the circuit back to state y. Then one test applies the input combination (y,x) to every k-th cell, and k tests are sufficient to apply it to every cell in the ILA. The following result [Friedman 1973, Sridhar and Hayes 1979] characterizes a C-testable ILA.

Theorem 8.2: An ILA is C-testable iff its equivalent sequential circuit has a reduced state table, and for every transition from any state y there exists a sequence that brings the circuit back to state y. □

The state diagram of a sequential circuit modeling a C-testable ILA is either a strongly connected graph, or it is composed of disjoint strongly connected graphs.

Now we discuss the generation of tests to check an entry (y,x) in the state table representing a C-testable ILA. To verify the next state $N(y,x)=p$, we use a *set of pairwise distinguishing sequences* [Sridhar and Hayes 1979]. A *pairwise distinguishing sequence* for a pair of states p and q, $DS(p,q)$, is an input sequence for which the output sequences produced by the circuit starting in states p and q are different. (Because the state table is assumed to be reduced, $DS(p,q)$ exists for every pair of states p and q). For example, for the state table of Figure 8.13, $DS(y_0,y_1)$=1 and $DS(y_1,y_2)$=01. A set of pairwise distinguishing sequences for a state p, $SDS(p)$, is a set of sequences such that for every state $q \neq p$, there exists a $DS(p,q) \in SDS(p)$. In other words, $SDS(p)$ contains sequences that distinguish between p and any other state. For the state table of Figure 8.13, $SDS(y_1)$={1,01}, where 1 serves both as $DS(y_1,y_0)$ and as $DS(y_1,y_3)$.

	x = 0	x = 1
y_0	y_1, 0	y_3, 1
y_1	y_1, 0	y_2, 0
y_2	y_0, 0	y_2, 0
y_3	y_0, 0	y_3, 1

Figure 8.13

The following procedure checks the entry (y,x). (Assume that the sequential circuit starts in state y.)

1. Apply x. This brings the circuit to state p.
2. Apply D_i, one of the sequences in $SDS(p)$. Assume D_i brings the circuit to state r_i.
3. Apply $T(r_i,y)$, a transfer sequence that brings the circuit from state r_i to state y (such a sequence always exists for a C-testable ILA).
4. Reapply the sequence $I_i=(x,D_i,T(r_i,y))$ as many times as necessary to have an input applied to every cell in the ILA. This step completes one test vector t_i.

Let k_i be the length of I_i. The test t_i applies (y,x) to every k_i-th cell of the ILA.

5. Apply k_i-1 shifted versions of t_i to the array.
6. Repeat steps 1 through 5 for every other sequence D_i in $SDS(p)$.

Example 8.6: We illustrate the above procedure by deriving tests to check the entry $(y_0,0)$ of the state table of Figure 8.13 modeling a C-testable ILA. Let $p=N(y_0,0)=y_1$ and $SDS(y_1)=\{1,01\}$. The first group of tests use $D_1=1$, $r_1=y_2$, and $T(y_2,y_0)=0$. Hence $I_1=(010)$ and the first test is $t_1=y_0$ $(010)^*$, where the notation $(S)^*$ denotes $SSS...$ (y_0 is applied to the first cell). Since $k_1=3$, we apply two shifted versions of t_1 to the ILA; we should also make sure that y_0 is applied to the first cell tested in each test. The two new tests are $y_20(010)^*$ and $y_210(010)^*$.

The second group of tests use $D_2=01$. Similarly we obtain $t_2=y_0(0010)^*$. Since $k_2=4$, we need three shifted versions of t_2 to complete this step. The seven obtained tests check the entry $(y_0,0)$ in all cells of the ILA, independent of the number of cells in the ILA. □

The test set for the entire ILA merges the tests for all the entries in the truth table of a cell.

Extensions and Applications

The ILAs we have considered so far are unilateral, one-dimensional, and composed of combinational cells with directly observable outputs. The fault model we have used assumes that the faulty cell remains a combinational circuit. In the following we will briefly mention some of the work based on more general assumptions.

The problem of testing *ILAs whose cells do not have directly observable outputs* (i.e., only the outputs of the last cell are POs) is studied in [Friedman 1973, Parthasarathy and Reddy 1981].

The testing of *two-dimensional ILAs*, whose structure is shown in Figure 8.14, is discussed in [Kautz 1967, Menon and Friedman 1971, Elhuni *et al.* 1986]. Shen and Ferguson [1984] analyze the design of VLSI multipliers constructed as two-dimensional arrays and show how these designs can be modified to become C-testable.

Cheng and Patel [1985] develop a pseudoexhaustive test set for a ripple-carry adder, assuming an *extended fault model* that also includes faults that transform a faulty cell into a sequential circuit with two states.

Testing of *bilateral ILAs*, where interconnections between cells go in both directions, is examined in [Sridhar and Hayes 1981b].

Sridhar and Hayes [1981a] have shown how the methods for testing combinational ILAs can be extended to *ILAs composed of sequential cells* having the structure given in Figure 8.15.

Sridhar and Hayes [1979, 1981a, 1981b] have studied the testing of *bit-sliced systems*, whose general structure is shown in Figure 8.16. Such a system performs operations on n-bit words by concatenating operations on k-bit words done by $N=n/k$ identical cells called *slices*. The operations are determined by s control lines distributed to

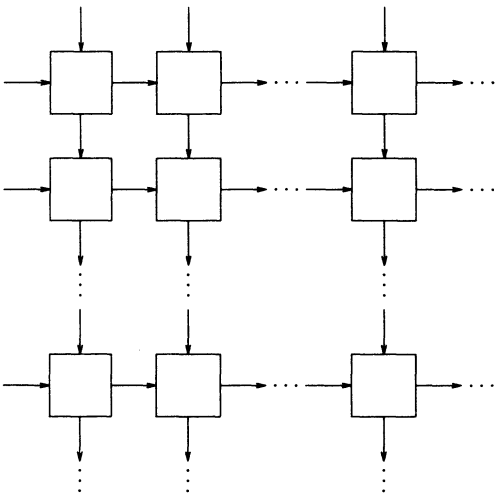


Figure 8.14 A two-dimensional ILA

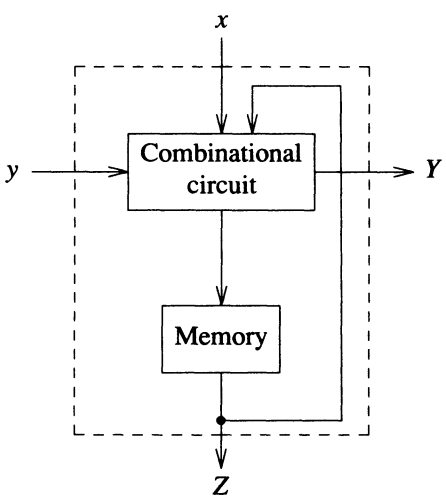


Figure 8.15

every slice. In general, a slice is a sequential device and the connections between adjacent slices are bilateral. This structure is typical for bit-sliced (micro)processors, where k is usually 2, 4, or 8. To model a bit-sliced system as an ILA, we can view it as composed of 2^s ILAs, each having a different basic cell implementing one of the 2^s possible operations.

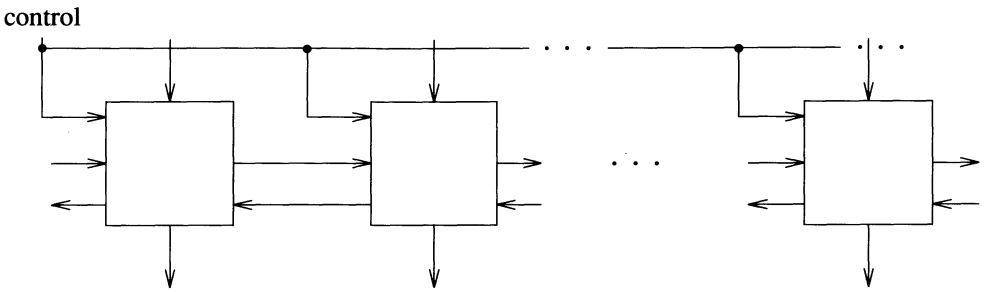


Figure 8.16 Bit-sliced processor

In practical bit-sliced systems, a slice is usually defined as an interconnection of small combinational and sequential modules. Figure 8.17 shows the 1-bit processor slice considered in [Sridhar and Hayes 1979, 1981a], which is similar to commercially available components. For this type of structure we apply a *hierarchical pseudoexhaustive testing approach.* First, we derive a pseudoexhaustive test T for a slice, such that T exhaustively tests every module. Because modules within a slice are small, it is feasible to test them exhaustively. The test T obtained for the slice of Figure 8.17 has 114 vectors. Second, we derive a pseudoexhaustive test T_A for the entire array, such that T_A applies T to every slice. A bit-sliced processor based on the slice of Figure 8.17 is C-testable and can be pseudoexhaustively tested with 114 vectors independent of its number of slices.

8.4 Functional Testing with Specific Fault Models

8.4.1 Functional Fault Models

Functional faults attempt to represent the effect of physical faults on the operation of a functionally modeled system. The set of functional faults should be *realistic*, in the sense that the faulty behavior induced by them should generally match the faulty behavior induced by physical faults. A functional fault model can be considered *good* if tests generated to detect the faults it defines also provide a high coverage for the SSFs in the detailed structural model of the system. (Because we do not know the comprehensiveness of a functional fault model, we cannot use the functional fault coverage of a test as a meaningful test quality measure.)

A functional fault model can be explicit or implicit. An *explicit model* identifies each fault individually, and every fault may become a target for test generation. To be useful, an explicit functional fault model should *define a reasonably small fault universe*, so that the test generation process will be computationally feasible. In contrast, an *implicit model* identifies classes of faults with "similar" properties, so that all faults in the same class can be detected by similar procedures. The advantage of an implicit fault model is that it does not require explicit enumeration of faults within a class.

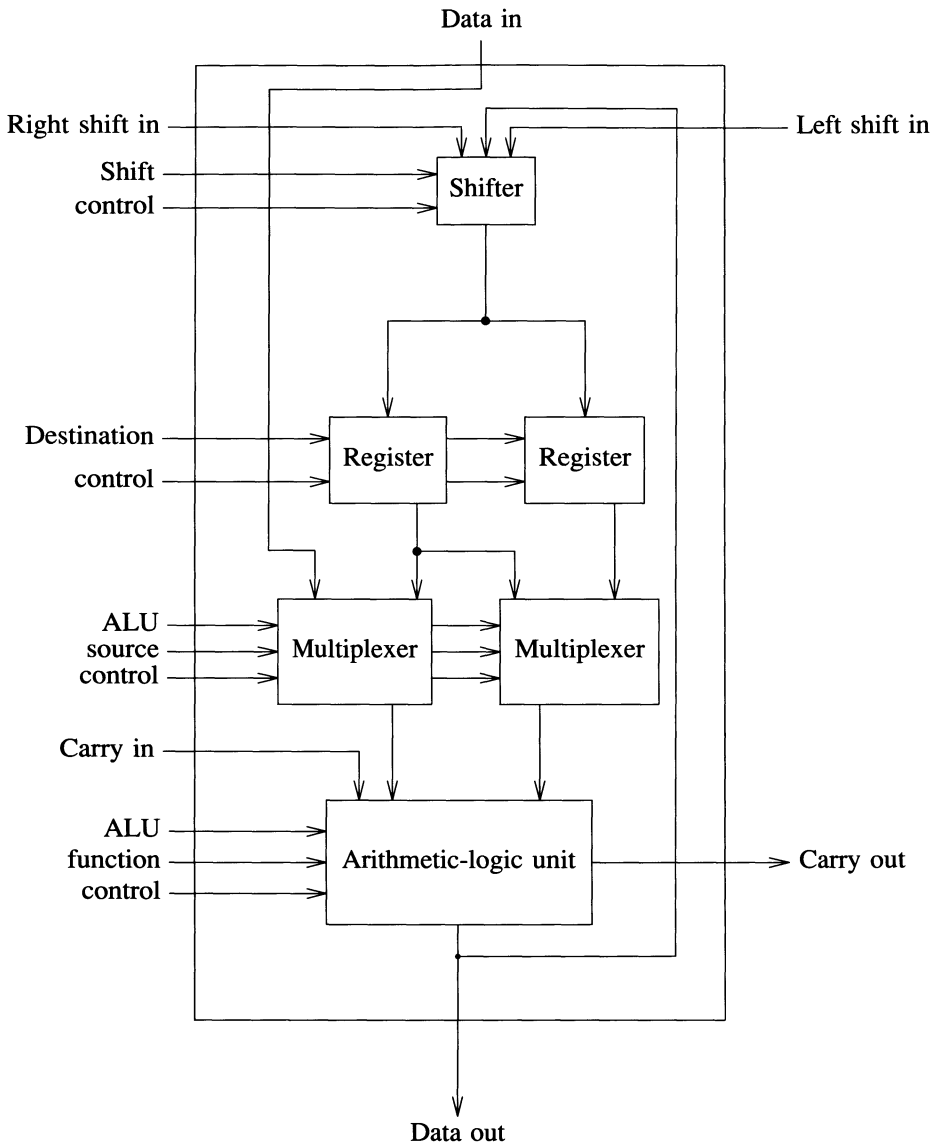


Figure 8.17 1-bit processor slice

Addressing Faults

Many operations in a digital system are based on decoding the address of a desired item. Typical examples include

- addressing a word in a memory;

- selecting a register according to a field in the instruction word of a processor;
- decoding an op-code to determine the instruction to be executed.

The common characteristic of these schemes is the use of an n-bit address to select one of 2^n possible items. *Functional addressing faults* represent the effects of physical faults in the hardware implementing the selection mechanism on the operation of the system. Whenever item i is to be selected, the presence of an addressing fault may lead to

- selecting no item,
- selecting item j instead of i,
- selecting item j in addition to i.

More generally, a set of items $\{j_1,j_2,...,j_k\}$ may be selected instead of, or in addition to, i.

An important feature of this fault model is that it forces the test generation process to check that the intended function is performed and also that no extraneous operations occur. This fundamental aspect of functional testing is often overlooked by heuristic methods.

Addressing faults used in developing implicit fault models for microprocessors are examined in the following section.

8.4.2 Fault Models for Microprocessors

In this section we introduce functional fault models for microprocessors. Test generation procedures using these fault models are presented in the next section.

Graph Model for Microprocessors

For functional test generation, a microprocessor can be modeled by a graph based on its architecture and instruction set [Thatte and Abraham 1980, Brahme and Abraham 1984]. Every user-accessible register is represented by a node in the graph. Two additional nodes, labeled IN and OUT, denote the connections between the microprocessor and the external world; typically, these are the data, address, and control busses connecting the microprocessor to memory and I/O devices. When the microprocessor is under test, the tester drives the IN node and observes the OUT node. A directed edge from node A to node B shows that there exists an instruction whose execution involves a transfer of information from node A to node B.

Example 8.7: Let us consider a hypothetical microprocessor with the following registers:

- A — accumulator,
- PC — program counter,
- SP — stack pointer holding the address of the top of a last-in first-out data stack,
- R1 — general-purpose register,
- R2 — scratch-pad register,

- SR — subroutine register to save return addresses (assume that no subroutine nesting is allowed),
- IX — index register.

Figure 8.18 shows the graph model for this microprocessor. The table of Figure 8.19 illustrates the mapping between some of the instructions of the microprocessor and the edges of the graph. (The notation (R) denotes the contents of the memory location addressed by register R.) Note that an edge may correspond to several instructions and an instruction may create several edges. □

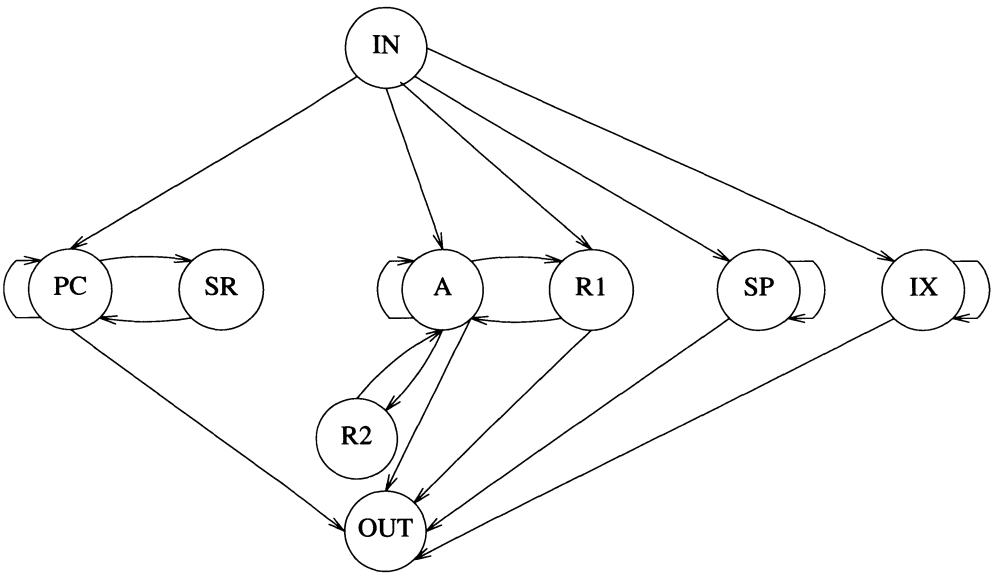


Figure 8.18 Graph model for a microprocessor

Fault Classes

Faults affecting the operation of a microprocessor can be divided into the following classes [Thatte and Abraham 1980]:

1. addressing faults affecting the register-decoding function;
2. addressing faults affecting the instruction-decoding and instruction-sequencing function;
3. faults in the data-storage function;
4. faults in the data-transfer function;
5. faults in the data-manipulation function.

Instruction	Operation	Edge(s)
MVI R, *a* where R∈ {A,R1,SP,IX}	R←*a*	IN→R
MOV Ra,Rb where Ra,Rb∈ {A,R1,R2}	Ra←Rb	Ra→Rb
ADD A,R1	A←A+R1	A→A R1→A
JMP *a*	PC←*a*	IN→PC PC→OUT
ADD A,(IX)	A←A+(IX)	IX→OUT IN→A A→A
CALL *a*	SR←PC PC←*a*	PC→SR IN→PC PC→OUT
RET	PC←SR	SR→PC PC→OUT
PUSH R where R∈ {A,R1}	SP←R SP←SP+1	SP→OUT R→OUT SP→SP
POP R where R∈ {A,R1}	SP←SP-1 R←(SP)	SP→SP SP→OUT IN→R
INCR R where R∈ {A,SP,IX}	R←R+1	R→R
MOV (IX), R where R∈ {A,R1}	(IX)←R	IX→OUT R→OUT

Figure 8.19 Instruction set

The overall fault model for the microprocessor allows for any number of faults in only one of these classes.

8.4.2.1 Fault Model for the Register-Decoding Function

Let us denote the decoding (selection) of a register R by a function f_D(R) whose fault-free value is R for any register R of the microprocessor. Whenever an instruction accesses a register R, an addressing fault affecting the register-decoding function leads to one of the following [Thatte and Abraham 1980]:

1. No register is accessed.
2. A set of registers (that may or may not include R) is accessed.

The first case is represented by $f_D(R)=\emptyset$, where $\emptyset$ denotes a nonexistent register. Here an instruction trying to write R will not change its contents, and an instruction trying to read R will retrieve a ONE or ZERO vector (depending on technology), independent of the contents of R; a ONE (ZERO) vector has all its bits set to 1(0).

In the second case, $f_D(R)$ represents the set of registers erroneously accessed. Here an instruction trying to write data d in R will write d in every register in $f_D(R)$, and an instruction trying to read R will retrieve the bit-wise AND or OR (depending on technology) of the contents of all registers in $f_D(R)$.

Example 8.8: For the microprocessor of Example 8.7, consider the fault $f_D(SR)=\emptyset$. The CALL a instruction will correctly jump to a, but it will not save the return address in SR. This fault will be detected by executing RET, since at that time a ZERO or ONE vector will be loaded into the PC.

With the fault $f_D(R2)=R1$, the instruction MOV R2,R1 will behave like NOP, and the instruction MOV A,R2 will result in transferring R1 into A. In the presence of the fault $f_D(R2) = \{R1,R2\}$, the instruction MOV R2,R1 executes correctly, but MOV A,R2 will transfer R1*R2 into A, where "*" denotes the bit-wise AND or OR operator. □

Note that a fault that causes $f_D(R_i)=R_j$ *and* $f_D(R_j)=R_i$ does not affect the correct operation (since it only relabels R_i and R_j); such a fault is undetectable.

8.4.2.2 Fault Model for the Instruction-Decoding and Instruction-Sequencing Function

Microprogramming Model for Instruction Execution

An instruction can be viewed as a *sequence of microinstructions*, where *every microinstruction consists of a set of microorders* which are executed in parallel. Microorders represent the elementary data-transfer and data-manipulation operations; i.e., they constitute the basic building blocks of the instruction set. This *microprogramming model* [Parthasarathy *et al.* 1982, Annaratone and Sami 1982, Brahme and Abraham 1984] is an abstract model, applicable regardless of whether the microprocessor is actually microprogrammed.

For example, the instruction ADD A,R1 can be viewed as a sequence of the following three microinstructions: (1) two (parallel) microorders to bring the contents of A and R1 to the ALU inputs, (2) an ADD microorder, and (3) a microorder to load the ALU result into A. The same instruction can also be considered as one microorder [Brahme and Abraham 1984]. We emphasize that the process of test generation (to be described in the next section) does not require the knowledge of the structure of instructions in terms of microinstructions and microorders.

Fault Model

The microprogramming model for the instruction execution allows us to define a comprehensive fault model for the instruction-decoding and instruction-sequencing function. Namely, addressing faults affecting the execution of an instruction I may cause one or more of the following fault effects [Brahme and Abraham 1984]:

1. One or more microorders are not activated by the microinstructions of I.

2. Microorders are erroneously activated by the microinstructions of *I*.
3. A different set of microinstructions is activated instead of, or in addition to, the microinstructions of *I*.

This fault model is general, as it allows for partial execution of instructions and for execution of "new" instructions, not present in the instruction set of the microprocessor.

A fault affecting an instruction *I* is *simple* if at most one microorder is erroneously activated during the execution of *I* (any number of microorders may be inactive). Thus there exists a one-to-one correspondence between the set of microorders and the set of simple faults.

Two microorders are said to be *independent* if neither of them modifies the source registers used by the other. Two simple faults are independent if the microorders they activate are independent. Our fault model allows for any number of simple faults, provided that they are (pairwise) independent. An example of a fault not included in this fault model is a fault that erroneously activates the sequence of microorders (ADD A,R1 ; MOV R2,A); such a fault is said to be *linked.*

8.4.2.3 Fault Model for the Data-Storage Function

The fault model for the data-storage function [Thatte and Abraham 1980] is a straightforward extension of the stuck-fault model. It allows any register in the microprocessor to have any number of bits *s-a*-0 or *s-a*-1.

8.4.2.4 Fault Model for the Data-Transfer Function

The data-transfer function implements all the data transfers among the nodes of the graph model of a microprocessor ("transfer" means that data are moved without being modified). Recall that an edge from node *A* to node *B* may correspond to several instructions that cause a data transfer from *A* to *B*. Although the same hardware may implement (some of) these data transfers, to make our model independent of implementation, we assume that every instruction causing a transfer $A \rightarrow B$ defines a separate *logical transfer path* $A \rightarrow B$, and each such transfer path may be independently faulty. For the microprocessor of Example 8.7, this model implies that the IN→PC transfer caused by a JMP *a* instruction may be faulty, while the same transfer caused by CALL *a* can be fault-free.

The fault model for the data-transfer function [Thatte and Abraham 1980] assumes that any line in a transfer path can be *s-a*-0 or *s-a*-1, and any two lines in a transfer path may be shorted.

8.4.2.5 Fault Model for the Data-Manipulation Function

The data-manipulation function involves instructions that change data, such as arithmetic and logic operations, register incrementing or decrementing, etc. It is practically impossible to establish a meaningful functional fault model for the data-manipulation function without any knowledge of the structure of the ALU or of the other functional units involved (shifter, incrementer, etc.). The usual approach in functional test generation is to assume that tests for the data-manipulation function are developed by some other techniques, and to provide means to apply these tests and observe their results. For example, if we know that the ALU is implemented as an

ILA, then we can derive a pseudoexhaustive test set for it using the methods described in Section 8.3.3. If the data-transfer function is checked first, this test set can be applied by loading the required operands in registers, executing the corresponding arithmetic or logic operations, and reading out the results. The data paths providing operands to the ALU and transferring the result from the ALU are checked together with the ALU.

8.4.3 Test Generation Procedures

In this section we examine test generation procedures for the functional fault models introduced in the previous section. These fault models are implicit and the test generation procedures deal with classes of faults without identifying the individual members of a class.

8.4.3.1 Testing the Register-Decoding Function

Because faults causing $f_D(R_i)=R_j$ and $f_D(R_j)=R_i$ are undetectable, our goal in testing the register-decoding function is to check that for every register R_i of the microprocessor, the size of the set $f_D(R_i)$ is 1 [Thatte and Abraham 1980]. This guarantees that none of the detectable addressing faults affecting the register-decoding function is present.

The testing procedure involves writing and reading of registers. For every register R_i, we predetermine a sequence of instructions WRITE(R_i) that transfers data from the IN node to R_i, and a sequence READ(R_i) that transfers the contents of R_i to the OUT node. Whenever there exist several ways to write or read a register, we select the shortest sequence.

Example 8.9: For the microprocessor of Example 8.7, we have the following sequences:

WRITE(A) = (MVI A,*a*)
READ(A) = (MOV (IX), A)

WRITE(R2) = (MVI A,*a*; MOV R2, A)
READ(R2) = (MOV A, R2; MOV (IX), A)

WRITE(SR) = (JMP *a*; CALL *b*)
READ(SR) = (RET)

□

With every register R_i we associate a *label* $l(R_i)$, which is the length of the sequence READ(R_i); here $l(R_i)$ represents the shortest "distance" from R_i to the OUT node. Using the READ sequences of Example 8.9, l(A)=l(SR)=1 and l(R2)=2.

The strategy of the testing procedure is to construct gradually a set A of registers such that

1. $f_D(R_i)\neq\emptyset$, for every $R_i\in A$
2. $f_D(R_i)\cap f_D(R_j)=\emptyset$, for every $R_i,R_j\in A$.

Eventually, A will contain all the registers of the microprocessor, and then these conditions will imply that $|f_D(R_i)|=1$ for every R_i.

Figure 8.20 presents the testing procedure (*Decode_regs*) to detect addressing faults affecting the register-decoding function. This procedure is executed by an external

tester that supplies the instructions for WRITE and READ sequences and checks the data retrieved by READ sequences. According to the start-small principle, registers are added to A in increasing order of their labels. The same ordering is used for reading out the registers in A.

```
Decode_regs()
begin
   A=∅
   add a register with label 1 to A
   for every register R_i not in A
      begin
         for Data=ZERO, ONE
            begin
               for every register R_j∈A  WRITE(R_j) with Data
               WRITE(R_i) with Data
               for every register R_j∈A  READ(R_j)
               READ(R_i)
            end
         add R_i to A
      end
end
```

Figure 8.20 Testing the register-decoding function

Theorem 8.3: If procedure *Decode_regs* executes without detecting any errors (in the data retrieved by READ sequences), then the register-decoding function is free of (detectable) addressing faults.

Proof: We will prove that successful completion of *Decode_regs* shows that $|f_D(R_i)|=1$ for every register R_i. The proof is by induction. Let R_i be a register with the lowest label among the registers currently not in A. Let us assume that all the registers currently in A have nonempty and disjoint sets $f_D(R_j)$. We want to show that this remains true after we add R_i to A.

If $f(R_i)=\emptyset$, READ(R_i) returns ZERO or ONE. Since READ(R_i) is executed twice, once expecting ZERO and once expecting ONE, one of them will detect the error. Hence if both READ(R_i) sequences execute without detecting errors, then $f(R_i)\neq\emptyset$.

If for some $R_j\in A$, $f_D(R_j)\cap f_D(R_i)\neq\emptyset$, then WRITE($R_j$) writes *Data* into a register whose contents are later changed to $\overline{Data}$ by WRITE(R_i). This error will be detected by one of the two READ(R_j) sequences. Hence if both READ(R_j) sequences execute without detecting errors, then $f_D(R_j)\cap f_D(R_i)=\emptyset$. Note that since $l(R_i)\geq l(R_j)$ for every $R_j\in A$, reading of R_j does not require routing of R_j through R_i.

The basis of induction, for the initial situation when A contains only one register, can be proved by similar arguments. Eventually, A will contain all the registers of the microprocessor. Then the relations $f_D(R_i)\neq\emptyset$ and $f_D(R_i)\cap f_D(R_j)\neq\emptyset$ imply that

$|f_D(R_i)| = 1$ for every R_i. Therefore successful completion of procedure *Decode_regs* shows that the register-decoding function is free of (detectable) addressing faults. □

Let n_R be the number of registers. The number of WRITE and READ sequences generated by procedure *Decode_regs* is proportional to n_R^2. Thus if all registers can be directly written and read out, i.e., every WRITE and READ sequence consists of only one instruction, then the number of instructions in the generated test sequence is also proportional to n_R^2. For an architecture with "deeply buried" registers, the worst-case length of the test sequence approaches n_R^3.

8.4.3.2 Testing the Instruction-Decoding and Instruction-Sequencing Function

Our goal is to detect all simple faults affecting the execution of any instruction. For this we have to ensure that any simple fault affecting an instruction I causes errors either in the data transferred to the OUT node or in a register that can be read after I is executed. This should be true if microorders of I are not activated and/or if additional (independent) microorders are erroneously activated. Missing microorders are generally easy to detect, as any instruction that does not activate all its microorders can be easily made to produce an incorrect result. To detect the execution of additional microorders, we associate different data patterns, called *codewords*, with different registers of the microprocessor [Abraham and Parker 1981, Brahme and Abraham 1984]. Let cw_i denote the codeword associated with register R_i. The set of codewords should satisfy the property that *any single microorder operating on codewords should either produce a noncodeword, or load a register R_i with a codeword cw_j of a different register.*

For n_R registers each having n bits, a set of codewords satisfying the above property can be obtained using a p-out-of-n code, where each codeword has exactly p bits set to 1, and $\binom{n}{p} \geq n_R$ (see Problem 8.10). Figure 8.21 illustrates a 5-out-of-8 codeword set.

cw_1	0 1 1 0 1 1 1 0
cw_2	1 0 0 1 1 1 1 0
cw_3	0 1 1 0 1 1 0 1
cw_4	1 0 0 1 1 1 0 1
cw_5	0 1 1 0 1 0 1 1
cw_6	1 0 0 1 1 0 1 1
cw_7	0 1 1 0 0 1 1 1
cw_8	1 0 0 1 0 1 1 1

Figure 8.21 A 5-out-of-8 codeword set

Example 8.10: Consider eight registers R1...R8, loaded with the codewords given in Figure 8.21. We will illustrate how several fault-activated microorders produce noncodewords:

1. ADD R1,R3 results in R1 having the noncodeword 11011011.

2. EXCHANGE R5,R7 results in both R5 and R7 having incorrect codewords.
3. OR R7,R8 produces the noncodeword 11110111 in R7.

Note that operations performed on noncodewords may produce a codeword. For example, if R4 and R5 have the noncodewords 00010101 and 10011001, then OR R4,R5 results in R4 having its correct codeword. □

If all registers are loaded with the proper codewords, simple faults affecting the execution of an instruction I will either cause an incorrect result of I or cause a register to have a noncodeword or the codeword of a different register. Hence to detect these faults, all registers must be read out after executing I. For a register that cannot be read directly, its READ sequence should be *nondestructive*, i.e., it should not change the contents of any other register (with the possible exception of the program counter).

Example 8.11: For the microprocessor of Example 8.7, the READ(R2) sequence used in the previous section — (MOV A,R2 ; MOV (IX),A) — destroys the contents of A. A nondestructive READ(R2), which saves and restores A, is

READ(R2)=(PUSH A; MOV A,R2; MOV (IX),A; POP A)

□

According to the start-small principle, we will check the READ sequences before checking the other instructions. We classify the faults affecting READ sequences according to the type of microorder they activate:

- type 1: microorders operating on one register, for example, increment, negate, or rotate;
- type 2: microorders causing a data transfer between two registers, for example, move or exchange;
- type 3: microorders executing arithmetic or logic operations on two source registers, for example, add.

Let S_1 be the set of registers modified by microorders of type 1. Let S_2 be the set of register pairs (R_j,R_k) involved in microorders of type 2, where R_k and R_j are, respectively, the source and the destination registers. Let S_3 be the set of register triples (R_j,R_k,R_l) involved in microorders of type 3, where R_k and R_l are the two source registers, and R_j is the destination register.

Figures 8.22, 8.23, and 8.24 present the testing procedures (*Read1*, *Read2*, and *Read3*) to detect, respectively, faults of type 1, type 2, and type 3 affecting the execution of READ sequences. Each procedure starts by loading codewords in registers. Since at this point we cannot assume that WRITE sequences are fault-free, the testing procedures take into account that some registers may not contain their codewords.

Theorem 8.4: If procedures *Read1*, *Read2*, and *Read3* execute without detecting any errors, then all READ sequences are fault-free.

Proof: We will prove only that procedure *Read3* detects all faults of type 3 affecting READ sequences. The proofs for procedures *Read1* and *Read2* are similar.

Consider a fault of type 3 that, during the execution of READ(R_i), erroneously activates a microorder that uses R_k and R_l as source registers and modifies R_j. If R_k and R_l now have their proper codewords, the first READ(R_j) detects a noncodeword

```
Read1()
begin
   for every R_i  WRITE(R_i) with cw_i
   for every R_i
      for every R_j ∈ S_1
         begin
            READ (R_i)
            READ (R_j)
            READ (R_i)
            READ (R_j)
         end
end
```

Figure 8.22 Testing for type 1 faults

```
Read2()
begin
   for every R_i  WRITE(R_i) with cw_i
   for every R_i
      for every (R_j,R_k) ∈ S_2
         begin
            READ (R_i)
            READ (R_j)
            READ (R_k)
            READ (R_i)
            READ (R_j)
         end
end
```

Figure 8.23 Testing for type 2 faults

(or an incorrect codeword) in R_j. However, if the microorder activated by READ(R_i) operates on incorrect data in R_k and/or R_l, then it may produce the correct codeword in R_j, so the first READ(R_j) does not detect an error. Next, READ(R_k) checks the contents of R_k; but even if R_k was not loaded with its codeword, it is possible that the first READ(R_j) changed R_k to its correct value. Similarly, READ(R_l) either detects an error or shows that R_l has its correct value. But READ(R_l) may change the contents of R_k. If the next READ(R_k) and READ(R_l) do not detect errors, we can be sure that both R_k and R_l have now their correct codewords. Thus when the second READ(R_i) is executed, the fault-activated microorder produces incorrect data in R_j, and the second READ(R_j) detects this error. Therefore, if no errors are detected, we can conclude that

```
Read3()
begin
   for every R_i  WRITE(R_i) with cw_i
   for every R_i
      for every (R_j,R_k,R_l) ∈ S_3
         begin
            READ (R_i)
            READ (R_j)
            READ (R_k)
            READ (R_l)
            READ (R_k)
            READ (R_l)
            READ (R_i)
            READ (R_j)
         end
end
```

Figure 8.24 Testing for type 3 faults

READ(R_i) is free of faults of type 3. As *Read3* repeats this test for every R_i, eventually all faults of type 3 affecting READ sequences are detected. □

We have assumed that all simple faults are independent. Brahme and Abraham [1984] present extensions of the *Read* procedures to detect linked faults as well.

The *Read* procedures also detect some faults affecting WRITE sequences. Figure 8.25 outlines a testing procedure (*Load*) to detect all faults affecting WRITE sequences. *Load* assumes that READ sequences are fault-free.

Procedure *Instr*, given in Figure 8.26, checks for faults affecting the execution of every instruction in the instruction set of the microprocessor. *Instr* assumes that both WRITE and READ sequences are fault-free. The question of whether every instruction should be tested for every addressing mode is answered depending on the orthogonality of the instruction set.

The complete test sequence for the instruction-decoding and instruction-sequencing function of the microprocessor is obtained by executing the sequence of procedures *Read1*, *Read2*, *Read3*, *Load*, and *Instr*.

The worst-case length of the test sequence checking the READ sequences is proportional to n_R^4, and the length of the test sequence checking the execution of every instruction is proportional to $n_R n_I$, where n_I is the number of instructions in the instruction set.

```
Load()
begin
   for every R_i WRITE(R_i) with cw_i
   for every R_i
      for every R_j
         begin
            READ(R_j)
            WRITE(R_i) with cw_i
            READ(R_j)
         end
end
```

Figure 8.25 Testing WRITE sequences

```
Instr()
begin
   for every instruction I
      begin
         for every R_i  WRITE(R_i) with cw_i
         execute I
         for every R_i  READ(R_i)
      end
end
```

Figure 8.26 Testing all instructions

8.4.3.3 Testing the Data-Storage and Data-Transfer Functions

The data-storage and data-transfer functions are tested together, because a test that detects stuck faults on lines of a transfer path $A \rightarrow B$, also detects stuck faults in the registers corresponding to the nodes A and B.

Consider a sequence of instructions that activates a sequence of data transfers starting at the IN node and ending at the OUT node. We refer to such a sequence as an *IN/OUT transfer*. For the microprocessor of Example 8.7, the sequence (MVI A,*a*; MOV R1,A; MOV(IX),R1) is an IN/OUT transfer that moves data a along the path IN→A→R1→OUT.

A test for the transfer paths and the registers involved in an IN/OUT transfer consists of repeating the IN/OUT transfer for different data patterns, so that

1. Every bit in a transfer path is set to both 0 and 1.

2. Every pair of bits is set to complementary values [Thatte and Abraham 1980].

Figure 8.27 shows a set of 8-bit data patterns satisfying these requirements.

1	1	1	1	1	1	1	1
1	1	1	1	0	0	0	0
1	1	0	0	1	1	0	0
1	0	1	0	1	0	1	0
0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1
0	1	0	1	0	1	0	1

Figure 8.27 Data patterns for checking an 8-bit transfer path

Clearly, for every transfer path involved in an IN/OUT transfer, a test sequence constructed in this way detects all the stuck faults on the lines of the transfer path and all the shorts between any two of its lines. The complete test for the data-storage and data-transfer functions consists of a set of IN/OUT transfers, such that every transfer path of the microprocessor is involved in at least one IN/OUT transfer.

8.4.4 A Case Study

Thatte and Abraham [1980] derived test sequences for an 8-bit microprocessor, based on functional fault models and testing procedures of the type presented in this chapter. These test sequences, consisting of about 9000 instructions, were evaluated by fault simulation using a gate and flip-flop model of the microprocessor. A sample of about 2200 SSFs were simulated. The obtained fault coverage was 96 percent, which is an encouraging result.

8.5 Concluding Remarks

Functional testing methods attempt to reduce the complexity of the test generation problem by approaching it at higher levels of abstraction. However, functional testing has not yet achieved the level of maturity and the success of structural testing methods. In this section we review some of the limitations and difficulties encountered in functional testing.

Although binary decision diagrams provide functional models that are easily used for test generation, their applicability in modeling complex systems has not been proven.

Pseudoexhaustive testing is best suited for circuits having a regular ILA-type structure. For arbitrary combinational circuits where at least one PO depends on many PIs, the number of tests required for pseudoexhaustive testing becomes prohibitive. Partitioning techniques can reduce the number of tests, but they rely on knowledge of the internal structure of the circuit, and their applicability to large circuits is hindered by the lack of good partitioning algorithms.

Explicit functional fault models are likely to produce a prohibitively large set of target faults. Implicit functional fault models have been successfully used in testing RAMs (see [Abadir and Reghbati 1983] for a survey) and in testing programmable devices such as microprocessors, for which test patterns can be developed as sequences of instructions. The test generation process is based on the architecture and the instruction set of a microprocessor and produces test procedures that detect classes of faults without requiring explicit fault enumeration. This process has not been automated and cannot generate tests for the data manipulation function.

Functional testing is an actively researched area. Some other approaches not examined in this chapter are presented in [Lai and Siewiorek 1983], [Robach and Saucier 1980], [Shen and Su 1984], [Lin and Su 1985], [Su and Hsieh 1981], and [Renous *et al.* 1989].

In general, functional testing methods are tightly coupled with specific functional modeling techniques. Thus the applicability of a functional testing method is limited to systems described via a particular modeling technique. Because there exist many widely different functional modeling techniques, it is unlikely that a generally applicable functional testing method can be developed. Moreover, deriving the functional model used in test generation is often a manual, time-consuming and error-prone process [Bottorff 1981].

Another major problem in functional testing is the lack of means for evaluating the effectiveness of test sequences at the functional level.

REFERENCES

[Abadir and Reghbati 1983] M. S. Abadir and H. K. Reghbati, "Functional Testing of Semiconductor Random Access Memories," *Computing Surveys*, Vol. 15, No. 3, pp. 175-198, September, 1983.

[Abadir and Reghbati 1984] M. S. Abadir and H. K. Reghbati, "Test Generation for LSI: A Case Study," *Proc. 21st Design Automation Conf.*, pp. 180-195, June, 1984.

[Abraham and Parker 1981] J. A. Abraham and K. P. Parker, "Practical Microprocessor Testing: Open and Closed Loop Approaches," *Proc. COMPCON Spring 1981*, pp. 308-311, February, 1981.

[Akers 1978] S. B. Akers, "Functional Testing With Binary Decision Diagrams," *Journal of Design Automation & Fault-Tolerant Computing*, Vol. 2, pp. 311-331, October, 1978.

[Akers 1985] S. B. Akers, "On the Use of Linear Sums in Exhaustive Testing," *Digest of Papers 15th Annual Intn'l. Symp. on Fault-Tolerant Computing*, pp. 148-153, June, 1985.

[Annaratone and Sami 1982] M. A. Annaratone and M. G. Sami, "An Approach to Functional Testing of Microprocessors," *Digest of Papers 12th Annual Intn'l. Symp. on Fault-Tolerant Computing*, pp. 158-164, June, 1982.

[Brahme and Abraham 1984] D. Brahme and J. A. Abraham, "Functional Testing of Microprocessors," *IEEE Trans. on Computers*, Vol. C-33, No. 6, pp. 475-485, June, 1984.

[Barzilai *et al.* 1981] Z. Barzilai, J. Savir, G. Markowsky, and M. G. Smith, "The Weighted Syndrome Sums Approach to VLSI Testing," *IEEE Trans. on Computers*, Vol. C-30, No. 12, pp. 996-1000, December, 1981.

[Bottorff 1981] P. S. Bottorff, "Functional Testing Folklore and Fact," *Digest of Papers 1981 Intn'l. Test Conf.*, pp. 463-464, October, 1981.

[Chang *et al.* 1986] H. P. Chang, W. A. Rogers, and J. A. Abraham, "Structured Functional Level Test Generation Using Binary Decision Diagrams," *Proc. Intn'l. Test Conf.*, pp. 97-104, September, 1986.

[Chen 1988] C. L. Chen, "Exhaustive Test Pattern Generation Using Cyclic Codes," *IEEE Trans. on Computers*, Vol. C-37, No. 2., pp. 225-228, February, 1988.

[Cheng and Patel 1985] W. T. Cheng and J. H. Patel, "A Shortest Length Test Sequence for Sequential-Fault Detection in Ripple Carry Adders," *Proc. Intn'l. Conf. on Computer-Aided Design*, pp. 71-73, November, 1985.

[Chiang and McCaskill 1976] A. C. L. Chiang and R. McCaskill, "Two New Approaches Simplify Testing of Microprocessors," *Electronics*, Vol. 49, No. 2, pp. 100-105, January, 1976.

[Elhuni *et al.* 1986] H. Elhuni, A. Vergis, and L. Kinney, "C-Testability of Two-Dimensional Iterative Arrays," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-5, No. 4, pp. 573-581, October, 1986.

[Friedman 1973] A. D. Friedman, "Easily Testable Iterative Systems," *IEEE Trans. on Computers*, Vol. C-22, No. 12, pp. 1061-1064, December, 1973.

[Friedman and Menon 1971] A. D. Friedman and P. R. Menon, *Fault Detection in Digital Circuits*, Prentice Hall, Englewood Cliffs, New Jersey, 1971.

[Kautz 1967] W. H. Kautz, "Testing for Faults in Cellular Logic Arrays," *Proc. 8th Symp. Switching and Automata Theory*, pp. 161-174, 1967.

[Lai and Siewiorek 1983] K. W. Lai and D. P. Siewiorek, "Functional Testing of Digital Systems," *Proc. 20th Design Automation Conf.*, pp. 207-213, June, 1983.

[Lin and Su 1985] T. Lin and S. Y. H. Su, "The *S*-Algorithm: A Promising Solution for Systematic Functional Test Generation," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-4, No. 3, pp. 250-263, July, 1985.

[McCluskey 1984] E. J. McCluskey, "Verification Testing — A Pseudoexhaustive Test Technique," *IEEE Trans. on Computers*, Vol. C-33, No. 6, pp. 541-546, June, 1984.

[McCluskey and Bozorgui-Nesbat 1981] E. J. McCluskey and S. Bozorgui-Nesbat, "Design for Autonomous Test," *IEEE Trans. on Computers*, Vol. C-30, No. 11, pp. 866-875, November, 1981.

[Menon and Friedman 1971] P. R. Menon and A. D. Friedman, "Fault Detection in Iterative Logic Arrays," *IEEE Trans. on Computers*, Vol. C-20, No. 5, pp. 524-535, May, 1971.

[Monachino 1982] M. Monachino, "Design Verification System for Large-Scale LSI Designs," *Proc. 19th Design Automation Conf.*, pp. 83-90, June, 1982.

[Moore 1956] E. F. Moore, "Gedanken Experiments on Sequential Machines," in *Automata Studies*, pp. 129-153, Princeton University Press, Princeton, New Jersey, 1956.

[Noon 1977] W. A. Noon, "A Design Verification and Logic Validation System," *Proc. 14th Design Automation Conf.*, pp. 362-368, June, 1977.

[Parthasarathy and Reddy 1981] R. Parthasarathy and S. M. Reddy, "A Testable Design of Iterative Logic Arrays," *IEEE Trans. on Computers*, Vol. C-30, No. 11, pp. 833-841, November, 1981.

[Parthasarathy *et al.* 1982] R. Parthasarathy, S. M. Reddy, and J. G. Kuhl, "A Testable Design of General Purpose Microprocessors," *Digest of Papers 12th Annual Intn'l. Symp. on Fault-Tolerant Computing*, pp. 117-124, June, 1982.

[Renous *et al.* 1989] R. Renous, G. M. Silberman, and I. Spillinger, "Whistle: A Workbench for Test Development of Library-Based Designs," *Computer*, Vol. 22, No. 4, pp. 27-41, April, 1989.

[Robach and Saucier 1980] C. Robach and G. Saucier, "Microprocessor Functional Testing," *Digest of Papers 1980 Test Conf.*, pp. 433-443, November, 1980.

[Shen and Ferguson 1984] J. P. Shen and F. J. Ferguson, "The Design of Easily Testable VLSI Array Multipliers," *IEEE Trans. on Computers*, Vol. C-33, No. 6, pp. 554-560, June, 1984.

[Shen and Su 1984] L. Shen and S. Y. H. Su, "A Functional Testing Method for Microprocessors," *Proc. 14th Intn'l. Conf. on Fault-Tolerant Computing*, pp. 212-218, June, 1984.

[Shperling and McCluskey 1987] I. Shperling and E. J. McCluskey, "Circuit Segmentation for Pseudo-Exhaustive Testing via Simulated Annealing," *Proc. Intn'l. Test Conf.*, pp. 58-66, September, 1987.

[Sridhar and Hayes 1979] T. Sridhar and J. P. Hayes, "Testing Bit-Sliced Microprocessors," *Digest of Papers 9th Annual Intn'l. Symp. on Fault-Tolerant Computing*, pp. 211-218, June, 1979.

[Sridhar and Hayes 1981a] T. Sridhar and J. P. Hayes, "A Functional Approach to Testing Bit-Sliced Microprocessors," *IEEE Trans. on Computers*, Vol. C-30, No. 8, pp. 563-571, August, 1981.

[Sridhar and Hayes 1981b] T. Sridhar and J. P. Hayes, "Design of Easily Testable Bit-Sliced Systems," *IEEE Trans. on Computers*, Vol. C-30, No. 11, pp. 842-854, November, 1981.

[Su and Hsieh 1981] S. Y. H. Su and Y. Hsieh, "Testing Functional Faults in Digital Systems Described by Register Transfer Language," *Digest of Papers 1981 Intn'l. Test Conf.*, pp. 447-457, October, 1981.

[Tang and Woo 1983] D. T. Tang and L. S. Woo, "Exhaustive Test Pattern Generation with Constant Weight Vectors," *IEEE Trans. on Computers*, Vol. C-32, No. 12, pp. 1145-1150, December, 1983.

[Thatte and Abraham 1980] S. M. Thatte and J. A. Abraham, "Test Generation for Microprocessors," *IEEE Trans. on Computers*, Vol. C-29, No. 6, pp. 429-441, June, 1980.

PROBLEMS

8.1 Show that a heuristic functional test that does not detect a detectable stuck fault on a PI of the circuit does not completely exercise its operation.

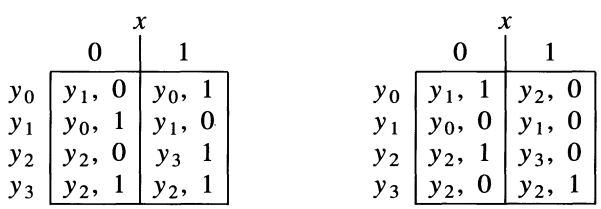
8.2 Generate all the feasible experiments for the binary decision diagram of Figure 8.6.

8.3 Using the binary decision diagram of Figure 8.6, generate a test in which f is sensitive to C.

8.4 Derive a pseudoexhaustive test set for the circuit of Figure 8.8(a) by partitioning it into the following three segments: (1) the subcircuit whose output is h, (2) gate y, and (3) gates g and x. Compare your results with those obtained in Example 8.4.

8.5 Show that any pseudoexhaustive test set based on a sensitized partitioning of a combinational circuit N detects all detectable SSFs in N.

8.6 Analyze the state tables shown in Figure 8.28 to determine whether the ILAs they represent are C-testable.



	x = 0	x = 1
y_0	y_1, 0	y_0, 1
y_1	y_0, 1	y_1, 0
y_2	y_2, 0	y_3 1
y_3	y_2, 1	y_2, 1

(a)

	x = 0	x = 1
y_0	y_1, 1	y_2, 0
y_1	y_0, 0	y_1, 0
y_2	y_2, 1	y_3, 0
y_3	y_2, 0	y_2, 1

(b)

Figure 8.28

8.7 Derive tests that verify the entry $(y_1,1)$ in all the cells of the ILA represented by the state table of Figure 8.13.

8.8 The ILA of Figure 8.29 is a realization of an n-bit parity function. Its basic cell implements an exclusive-OR function. Show that the ILA is C-testable with four tests.

(Because the cells do not have direct outputs, analyze fault propagation through a cell using propagation D-cubes.)

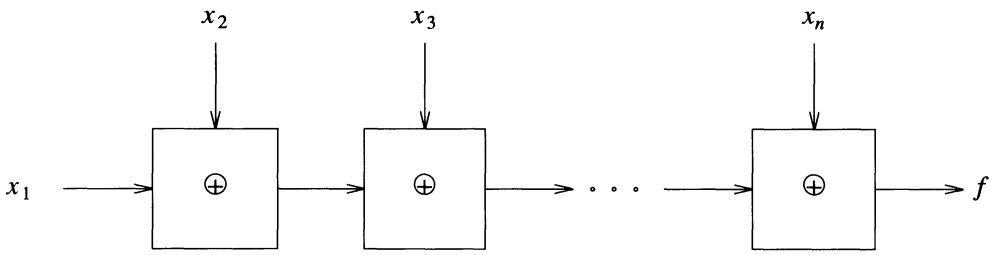


Figure 8.29

8.9 Consider a gate-level model of a 2-to-4 decoder. Show that any SSF leads to the following faulty behavior: for any input vector, instead of, or in addition to the expected output line, some other output is activated, or no output is activated.

8.10 Show that a p-out-of-n codeword set satisfies the required property of codewords with respect to any microorder of the form R1←R1*R2, where "*" denotes an ADD, AND, OR, or XOR operation.