TALLINN TECHNICAL UNIVERSITY
Faculty of Information Technology
Department of Computer Engineering
Chair of Computer Engineering and Diagnostics

THEORETICAL BACKGROUND FOR THE APPLET-BASED EXERCISES:

- **TEST GENERATION AND FAULT DIAGNOSIS**
- **BOUNDARY SCAN**

Natalja Mazurova
990851LAS

TALLINN 2003

TABLE OF CONTENTS

GLOSSARY DEFINITIONS

ALFSR - Autonomous LFSR
ASIC - Application-Specific Integrated Circuit
ATE - Automatic Test Equipment
ATPG - Automatic Test Pattern Generator
BIST - Built-In Self-Test
BSC - Boundary Scan Cell (in BST)
BSDL - Boundary Scan Description Language
BSR - Boundary Scan Register (in BST)
BST - Boundary Scan Test
CMOS - Complementary MOS
CUT - Circuit Under Test
DR - Data Register (in BST)
FAN - FAN-out-oriented test generation algorithm
FC - Fault Coverage
FEF - Functionally Equivalent Faults
GDF - Gate Delay Fault
IC - Integrated Circuit
IR - Instruction Register (in BST)
LFSR - Linear Feedback Shift Register
MOS - Metal-Oxide-Silicon
MUX - Multiplexer
NMOS - N-channel MOS
PDF - Path Delay Fault
PI - Primary Input
PO - Primary Output

PODEM - Path-Oriented DEcision-Making algorithm
RPR - Random Pattern-Resistant faults
RU - Replaceable Unit
SSA - Single Stuck-At
SSF - Single Stuck-at Faults
TAP - Test Access Port (in BST)
TCK - Test Clock (in BST)
TDI - Test Data Input (in BST)
TDO - Test Data Output (in BST)
TMS - Test Mode Select (in BST)
TRST - Test Reset (in BST)
UUT - Unit Under Test

1 FAULT MODELS AND FAULT SIMULATION

Introduction. Failure modes are manifested on the logical level as incorrect signal values. A fault is a model that represents the effect of a failure by means of the change that is produced in the system signal. Several defects are usually mapped to one fault model, and it is called a many-to-one mapping. However some defects may also be represented by more than one fault model. Fault models have the advantage of being a more tractable representation than physical failure modes. It is possible to mark most commonly used fault models.

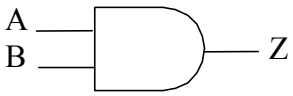
Fault Model	Description
Single stuck-at faults (SSA)	One line takes the value 0 or 1.
Multiply stuck-at faults	Two or more lines have fixed values, not necessarily the same.
Bridging faults	Two or more lines that are normally independent become electrically connected.
Delay faults	A fault is caused by delays in one or more paths in the circuit.
Intermittent faults	Caused by internal parameter degradation. Incorrect signal values occur for some but not all states of the circuit. Degradation is progressive until permanent failure occurs.
Transient faults	Incorrect signal values caused by coupled disturbances. Coupling may be via power bus capacitive or inductive coupling. Includes internal and external sources as well as particle irradiation.

Table 1. Most commonly used fault models

As a model, the fault does not have to be an exact representation of the defects, but rather, to be useful in detecting the defects. For example, the most common fault model assumes single stuck-at (SSA) lines even though it is clear that this model does not accurately represent all actual physical failures. The rational for continuing to use stuck-at-fault model is the fact that it has been satisfactory in the past. Also, test sets that have been generated for this fault type have been effective in detecting other types of faults. However, as with any model, a fault cannot represent all failures. Further will be discussed a bit closer the fault models that have been brought in Table 1.

1.1 MOST COMMON FAULTS MODELS

1.1.1 Single stuck-at faults. As it was mentioned earlier a single stuck-at (SSA) fault represents a line in the circuit that is fixed to logic value 0 or 1. Independent of how accurately the stuck-at fault represents the physical defect, we next continue investigating how to generate patterns that detect these faults. Lets examine the behavior, one the logical level, of a two-input AND gate when stuck-at faults are injected, one at time, on all input and output leads. This is illustrated in Figure 1.



Inputs AB	**Good response**	**Faulty response**					
		A/0	B/0	Z/0	A/1	B/1	Z/1
00	0	0	0	0	0	0	1
01	0	0	0	0	1	0	1
10	0	0	0	0	0	1	1
11	1	0	0	0	1	1	1

Figure 1. Stuck-at faults on a two-input AND gate and their detection

All input combinations are given in the first column of the table. The fault-free and faulty circuit's responses, R and R_f, respectively, are listed in the other columns of the table for each stuck-at fault. The fault is detected whenever there is an input combination such that $(R \text{ xor } R_f) = 1$. Stuck-at faults on line A are indicated by $A/0$ for

stuck-at 0 and *A/1* for stuck-at 1. Similar notations are used for the other lines. A careful observation of the table indicates that a faulty response of the circuit is not always observable. For instance, with *A/0*, it is expected that the output will always be zero irrespective of the input combination. Thus the faulty response differs from the fault-free response only when the input combination *AB = 11* is applied on the circuit. This combination is considered as a *test pattern* that detects the fault *A/0*. In a similar fashion, we can determine that this pattern also detects *B/0* and *Z/0*. This pattern detects any of the faults but does not help diagnose which fault actually occurred. We notice also that a fault may be detected by more than one pattern. This is the case with *Z/1*. Any of the three test patterns (*10)*, (*01)*, and (*00)* should be sufficient to detect the fault. The latter pattern (*00)* is the only one that determines that the failure is due to *Z/1*. If the main aim is to detect the failures rather than diagnose them, only three patterns are necessary to accomplish the task. They are (*01)*, (*10)*, and *(11)* and they form a test set of length 3.

Summing up the SSA faults for the two-input AND gate, we have shown that:

- Three patterns are sufficient to detect all faults
- The three faults *A/0*, *B/0*, and *Z/0* are equivalent
- Detecting stuck-at 1 faults on the inputs guarantees detection of the same fault on the output

1.1.2 Multiply stuck-at faults. A defect may cause multiply stuck-at faults. That is, more than one line may be stuck-at high or low simultaneously. With decreased device geometry and increased gate density on the chip, the likelihood is greater that more than one SSA fault can occur simultaneously. It has been recommended to check *m*-way stuck-at faults up to *m = 6* [Goldstein 1970]. This is particularly true with present technology circuits because of the high device density. A set of *m* lines has 2^m combinations of SA faults. Since the total number of *m*-sets of lines in a *N*-line circuit is

$$C(N,m) = \frac{N!}{m!(N-m)!}$$

the total number of *m*-way faults is $2^m C(N,m)$.

Thus the total number of multiply faults is:

$$\sum_{m=1}^{N} 2^m C(N,m) = 2^N - 1$$

In detecting multiply stuck-at faults, it is always possible to use exhaustive and pseudoexhaustive testing. However, this is not practical for large circuits. The most important factors that affect the detectability of multiply stuck-at faults are the number of primary outputs and the reconverging fanouts [Schertz 1971, Hughes 1984].

1.1.3 Bridging faults. Such faults occur when two or more lines are shorted together and create wired logic. When the fault involves *r* lines with $r \geq 2$, it is said to be of *multiplicity r*; it is a *simple bridging fault*. Multiply bridging faults are more likely to occur at the primary inputs of a chip. Bridging faults are becoming more predominant because the devices are becoming smaller and the gate density higher. The total number of all possible simple bridging faults in a *m*-line circuit is *C(m,2)*. However, in reality most pairs of lines are not likely to be shorted. Thus the actual number is much smaller than theoretically calculated and is layout dependent.

Bridging faults may cause a change in the functionality of the circuit that cannot be represented by a known fault model. An example of this type of fault is illustrated in Figure 2.

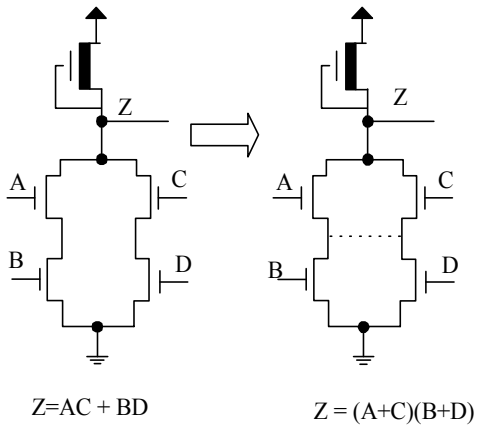


Figure 2. Change in functionality due to bridging faults

Here you can see that the function of the good NMOS circuit is *AC+BD*, while the bridging fault changes the functionality of the gate to *(A+C)(B+D)*. Also, the NOR gate has been transformed into a sequential circuit.

For detecting bridging faults have been used SSA fault test sets. They yield 100% fault detection for some special circuits. The approach is to alter the order of the patterns. It is also possible to use exhaustive test sets.

1.1.4 Delay faults. It is possible for a circuit to be structurally correct but to have signal paths with delays that exceed the bounds required for correct operation. In such a case, a *delay fault* is said to have occurred. The ultimate goal for detecting delay faults is to determine that the circuit works without malfunctions at the designed clock frequency. Thus it is appropriate to assume that the bound for correct operations should be the slack of the signal at the lead at which the fault is detected. The slack is difference between the clock period and the longest delay path.

Two main models are used for delay testing. The first, *gate delay fault* (GDF), is gate-oriented. The second model, *path delay fault* (PDF), is path-oriented.

The GDF model assumes that the delay faults are lumped at the faulty gate. The delay at the output of the gate will depend on whether this signal is switching from 0 to 1 (rise) or vice versa (fall). There is a disadvantage in adopting this type of fault because it does not capture the cumulative effects from other gates, and it also ignores the delays in the interconnect wires. Also, the gate delay may cause a local delay at its output without affecting the delay of the circuit.

The PDF model takes into account the cumulative delay from the primary input to the output. Although this model requires the consideration of too many paths, it is more realistic, particularly for present technology circuits, where delays are due primarily to the interconnect wires. As the technology features decreased, gate delays have been reduced. Meanwhile, the resistance of the interconnect wires has increased due to the reduction of their cross-sectional area. Except for domino logic gates, delay testing of CMOS circuits consists of applying a pair of input patterns at the desired operational speed and observing the outputs for early or late transitions.

1.1.5 Temporary faults. The last two types of faults, transient and intermittent faults, are made up the temporary faults. Table 2 lists these types of temporary faults and some of their main causes. These failures are much harder to track because it is usually not possible to reproduce the fault when a component, chip, or board is tested. They are encountered in different digital components, but are particularly important in memory chips and microprocessors.

Type	Causes
Transient	Power supply disturbances Electromigration interference Charged particles Atmospheric discharges Electrostatic discharges
Intermittent	Parameter degradation

Table 2. Temporary faults

A *transient* fault occurs when a logic signal has its value temporarily altered by noise signals, and the rest of the circuit may interpret the resulting signal incorrectly. Such a fault is difficult to diagnose and correct. It is thus important to minimize the noise in the circuit and increase the circuit's noise immunity.

Intermittent faults are recognized to be an important cause of field failures in computer systems. Very little is known about the failure mechanisms because spontaneous intermittent failures are difficult to observe and control.

1.2 FAULT SIMULATION CONCEPTS

1.2.1 Introduction to fault simulation. Fault simulation is performed during the design cycle to achieve the following goals:

- Testing specific faulty conditions
- Guiding the test pattern generator program
- Measuring the effectiveness of the test patterns
- Generating fault dictionaries

To perform its task, the fault simulation program requires, in addition to the circuit model, the stimuli, and the responses of a good circuit to the stimuli, a fault model and a fault list. This is illustrated in Figure 3.

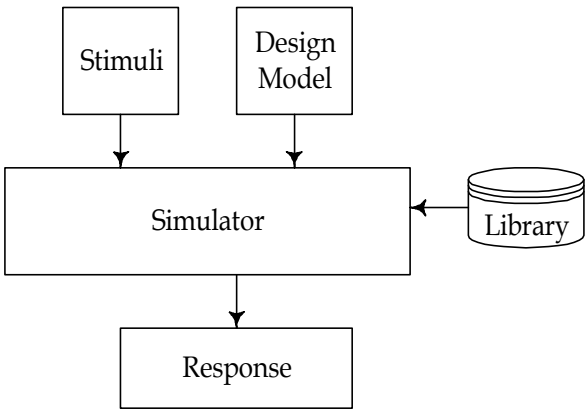


Figure 3. Elements of fault simulation

As was mentioned earlier, there are different fault models, and the most widely used is the stuck-at model. Test patterns generated for this model have proven to be useful for other types of models, such as multiply stuck-at, bridging, and delay faults. The responses deduced by the fault simulator are used to determine the fault coverage.

In Figure 3 is illustrated the fault simulation process where a fault is considered from the list and a pattern is applied to the circuit. If the fault is detected, it is dropped from the fault list and the next fault is considered. Otherwise, another pattern is applied, and if the fault is not detected when all patterns are applied, the fault is then considered undetectable by the test and is removed from the fault list. The process is continued until the fault list is empty.

1.2.2 Fault simulation results. The output of a fault simulator separates faults into several fault categories. If we can detect a fault at a location, it is a testable fault. A testable fault must be placed on a controllable net, so that we can change the logic level at that location from *0* to *1* and from *1* to *0*. A testable fault must also be on an observable net, so that we can see the effect of the fault at a primary output (PO). This means that uncontrollable nets and unobservable nets result in faults we cannot detect. We call these faults untested faults, untestable faults, or impossible faults.

If a PO of the good circuit is the opposite to that of the faulty circuit, we have a detected fault (sometimes called a hard-detected fault or a definitely detected fault). If the POs of the good circuit and faulty circuit are identical, we have an undetected fault. If a PO of the good circuit is a *1* or a *0* but the corresponding PO of the faulty circuit is an *X* (unknown, either *0* or *1*), we have a possibly detected fault (also called a possible-detected fault, potential fault, or potentially detected fault).

If the PO of the good circuit changes between a *1* and a *0* while the faulty circuit remains at *X*, then we have a soft-detected fault. Soft-detected faults are a subset of possibly detected faults. Some simulators keep track of these soft-detected faults separately. Soft-detected faults are likely to be detected on a real tester if this sequence occurs often. Most fault simulators allow you to set a fault-drop threshold so that the simulator will remove faults from further consideration after soft-detecting or possibly detecting them a specified number of times. This is called fault dropping (or fault discarding). The more often a fault is possibly detected, the more likely it is to be detected on a real tester. A redundant fault is a fault that makes no difference to the circuit operation. A combinational circuit with no such faults is irredundant.

1.2.3 Fault coverage. The effectiveness of the test sets is usually measured by the fault coverage (FC). This is the percentage of detectable faults in the circuit under test (CUT) that are detected by the test set. The fault coverage is defined as:

$$\textit{fault coverage} = \textit{faults detected} / \textit{total number of faults}$$

A more realistic expression can be found as:

$$\textit{fault coverage} = \textit{faults detected} / \textit{detectable faults}$$

The set is complete if its fault coverage is 100%. This level of fault coverage is desirable but rarely attainable in most practical circuits. Moreover, 100% fault coverage does not guarantee that the circuit is fault-free. The test checks only for failures that can be represented by the model used, such as a stuck-at-fault model that was mentioned earlier. Other failures are not necessarily detected.

2 TEST GENERATION

Introduction. In this introduction part we distinguish between various types of tests according to the test generation method.

The first type is the *exhaustive* test. Since a test pattern is a combination of the values applied on the primary inputs (PI) of a CUT, it is conceivable to use all possible combinations (an exhaustive test set) and apply them to the circuit. This exhaustive approach to testing has the advantage of being easy to generate and of yielding 100% fault coverage. However, such a testing method is efficient only for purely combinational small circuits.

An alternative to exhaustive testing is the *pseudoexhaustive* test. The approach is to test the components of a circuit exhaustively without having to apply an exhaustive test on the entire circuit. Figure 4 illustrates this approach quite well. The circuit has eight PIs. The length of an exhaustive test is 256 patterns. Instead, the circuit is partitioned into three subcircuits: α, β, and γ.

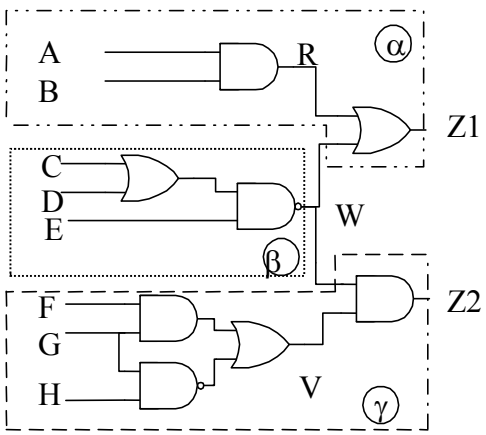


Figure 4. Verification testing

It is possible to test the first component exhaustively using four patterns and observing through the primary output, Z_1. For this, the other input to the OR gate should be controlled by zero. The other two components each depend on three of the inputs. They can be tested exhaustively with eight patterns each. The total length of the test is the sum of the three individual tests, 20 patterns instead of 256 patterns. To test subcircuit γ, we need to sensitize the response to the test through Z_2. This implies keeping $W = 1$. As for component β, the response may be sensitized through Z_1 or Z_2, and hence we need to keep $R = 0$ or $V = 1$. This type of testing requires an efficient way of partitioning the circuit.

Another special case of exhaustive testing is known as *verification testing*. It is applicable to circuits where each PO is a function of only a subset of PIs. The circuit we used for pseudoexhaustive testing has two POs, Z_1 and Z_2. Each output is dependent on a subset of the PIs, 5 and 6, respectively, as can be determined from Figure 5. The corresponding exhaustive test sets are of lengths 32 and 64. The length of the test for the circuit is then the sum of both test sets and is equal to 96 patterns. This is definitely a much shorter test set than the exhaustive test of length 256 patterns.

Test patterns may also be generated in random order. Lets take a quick look at the *pseudorandom* test. The cost of generating the test is minimal. A fault simulator is needed to grade the test and assess the fault coverage. The advantage of random testing is that it has been shown to detect a large percentage (possibly 85%) of stuck-at faults. Consequently, many commercial ATPGs use random testing as a first stage of the test pattern generation and then apply heuristics to deal with the still undetected faults, which are called *random pattern-resistant* (RPR) *faults*.

In purely random testing, a test pattern may be generated more than once. However, pseudorandom test generation is more appropriate to ensure that there is no repetition of patterns. Pseudorandom test sets may be generated by a software program or by a linear feedback shift register (LFSR). LFSR will be discussed later in the last subchapter.

Deterministic tests are fault-oriented tests. In this case, patterns are generated targeting a specific fault model.

2.1 BASIC OPERATIONS OF ATPG

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

To *provoke* or *excite* a line is to *control* it to a logic value that is the complement of the value at which it is stuck; this is equivalent to placing the faulty signal on the line. This signal is a discrepancy from the fault-free circuit. For example, to provoke the stuck-at 1 fault on line *W*, *W/1*, of the circuit in Figure 5, we must put *0* on this line, *W = 0*.

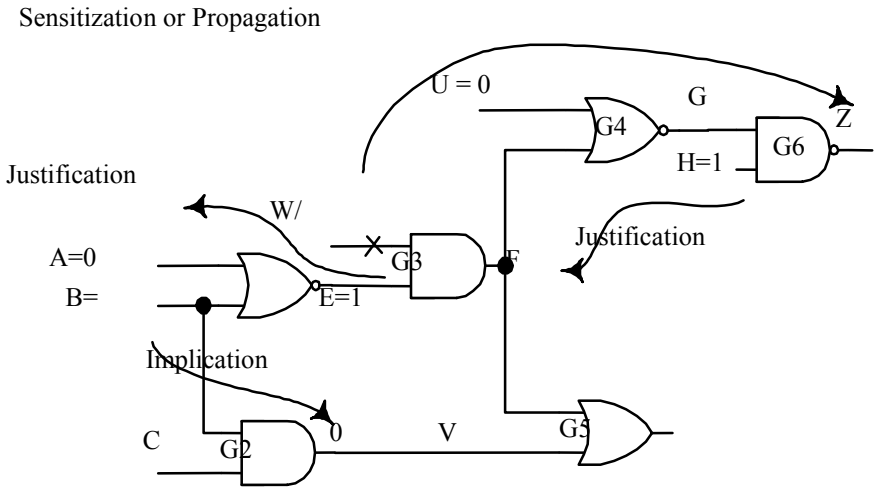


Figure 5. Test pattern generation terminology

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

2.2 PODEM ALGORITHM

The path-oriented decision-making (PODEM) algorithm starts the search for the test pattern at the PIs of the circuit. The algorithm is outlined by the flowchart in Figure 6.

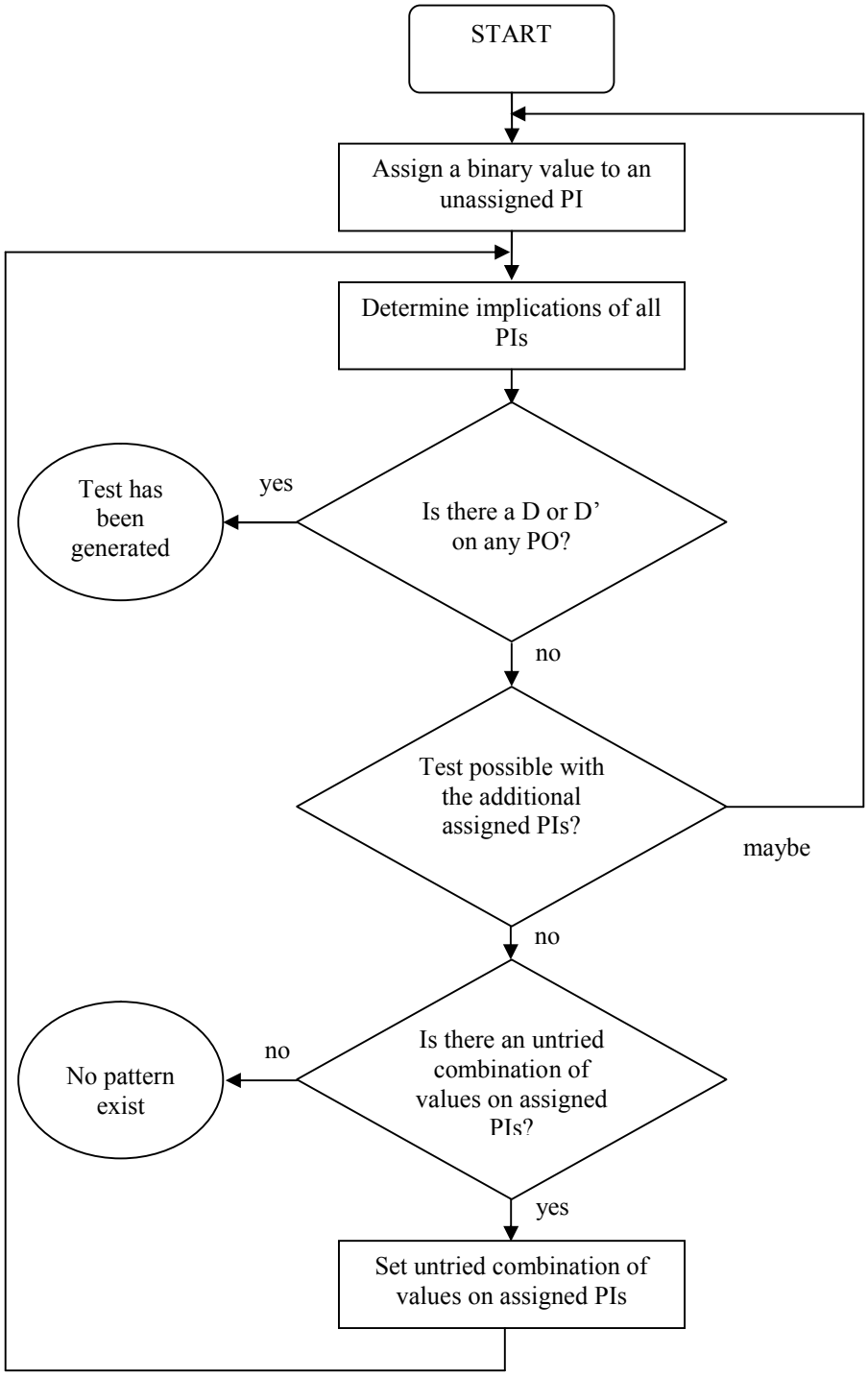


Figure 6. PODEM algorithm

Starting with an objective, a specific fault to be detected, a search tree is created in which two choices are available, *0* and *1*, for the PIs. The choice is random. Evaluate the implications of this choice on the subsequent gates to the output. If it furthers the objective – controlling the fault site to the intended value – accept it and select another PI. If an inconsistency occurs, the algorithm backtracks and selects another input combination. The search stops whenever a pattern generated or no patterns are possible (undetectable fault).

The advantage of PODEM is that it cuts down on the backtracking, as will be demonstrated using the next circuit in Figure 7.

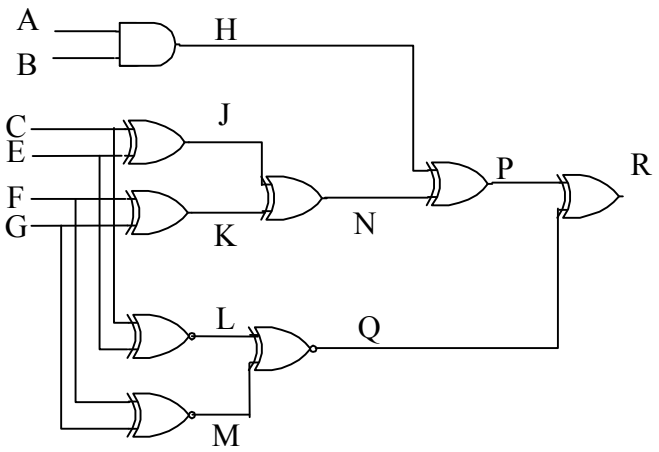


Figure 7. The circuit that uses reconvergent fanout with XOR gates

The target fault is *H/0* and the search tree is shown in Figure 8.

1. Assign *x* to all inputs.
2. Assigning *0* to the PI, *A*, causes $H = 0$; hence this choice does not further our goal and it is discarded. Then $A = 1$.
3. Similarly, $B = 0$ is rejected and $B = 1$ is selected.
4. We proceed with $C = 0$. The implication of this value is not furthering the objective, nor is it blocking it.
5. Thus we select $E = 0$; this then results in $J = 0$ and $L = 1$.
6. Similarly, we select $F = G = 0$, which implies that $K = 0$, $M = 1$, $N = 0$, $Q = 1$.
7. We can propagate *D* on *P*, then *D'* on the PO, *R*. (SA0, SA1 faults are denoted by *D*, *D'* correspondingly).

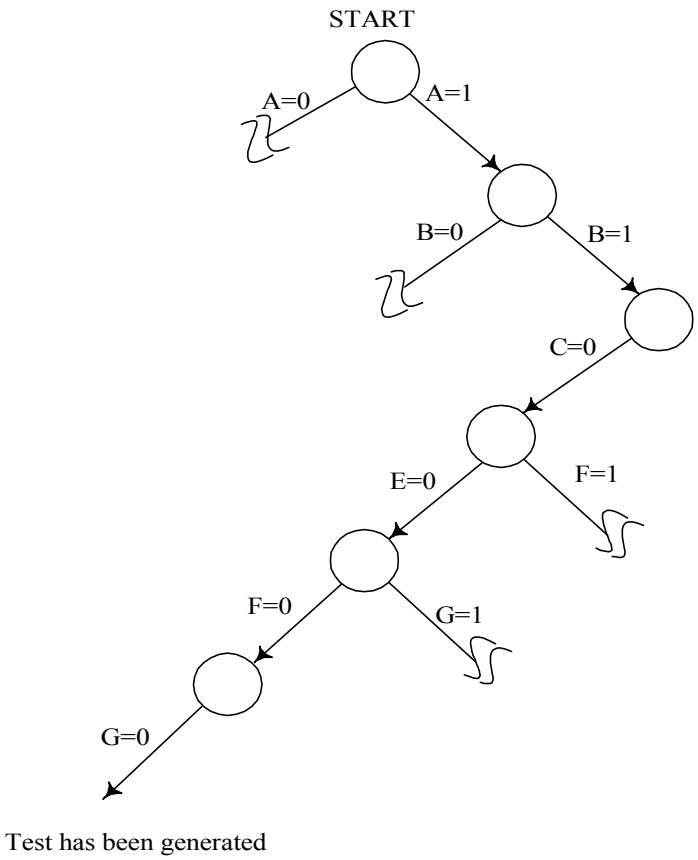


Figure 8. PODEM search tree for circuit in Figure 8

From the search tree in Figure 8, the primary inputs are then {ABCEFG} = {110000}.

2.3 FAN ALGORITHM

PODEM has been effective in reducing the occurrences of backtracking, but the new strategy has helped in reducing them even further: FAN [Fujiwara 1983]. This algorithm does extensive analysis of the circuit connectivities before backtracking. Also, the search is aided by testability analysis.

The fan-out-oriented test generation (FAN) algorithm remedies the exhaustive approach of PODEM by pruning from the search tree any branching that would not yield a solution.

FAN uses the following strategies [Fujiwara 1986]:

1. In each step of the algorithm, determine as many signal values as possible that can be implied uniquely.
2. Assign a faulty signal D or D' that is uniquely determined or implied by the fault in question.
3. When the D-frontier consists of a single gate, apply a unique sensitization.
4. Stop the backtrack at a headline, and postpone the line justification for the headline to later.
5. Multiply backtracking is more efficient than backtracking along a single path.
6. In the multiply backtrack, if an objective at a fanout point has a contradictory requirement, stop the backtrack so as to assign a binary value to the fan-out point.

Using these strategies, FAN minimized the backtracks and reduced the test generation time. In Table 3 you can see the comparison of FAN and PODEM for five benchmark circuits. The last two columns are of particular interest because they indicate the effectiveness of the algorithm. These columns give the number of faults that were not detected after 1000 backtracks; they are referred to as the aborted faults.

	Computing time		**Average backtracks**		**Percentage of faults aborted**	
Circuit	**PODEM**	**FAN**	**PODEM**	**FAN**	**PODEM**	**FAN**
1	1.3	1	4.9	1.2	0.32	0.37
2	3.6	1	42.3	15.2	2.26	3.13
3	5.6	1	61.9	0.6	2.42	4.00
4	1.9	1	5.0	0.2	0.99	1.10
5	4.8	1	53.0	23.2	0.82	1.02

Table 3. Comparison of PODEM and FAN algorithms

2.4 LFSR

In random test pattern generation, a pattern may be repeated several times in the process. However, using pseudorandom yields random patterns without repetition. This is equivalent to selection without replacement. The length of the test generated in such a manner depends on the seed of the random number generator. It is conceivable to use

some software tools to generate the random patterns and then store them in a ROM that is placed on the chip. One of the types of hardware may be used instead, and it is a linear feedback shift register (LFSR). Example of LFSR is shown in Figure 9.

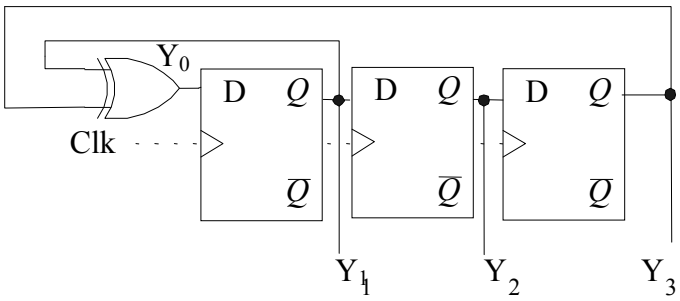


Figure 9. ALFSF for pseudorandom test pattern generation

As its name implies, the LFSR is a shift register with feedback from the last stage and other stages. Besides the clock, it has no other inputs. This is why it is also referred to as *autonomous LFSR* (ALFSR). The outputs of the flip-flops form the test pattern. For this example, there are 3-bit test patterns. In Figure 9, the parity of all feedback leads is the input to the register. Thus we have $Y_0 = (Y_1 \text{ xor } Y_3)$, $Y_1 = y_0$, $Y_2 = y_1$, and $Y_3 = y_2$, where $y_1y_2y_3$ represents the present state and $Y_1Y_2Y_3$ is the next state of the register. The LFSR was arbitrarily initialized to *001*, which is the first pattern appearing on the LFSR. Next we clock the circuits as many times as it is necessary to reproduce this first pattern as illustrated in Table 4. In all, there are seven patterns, since all-zeros pattern cannot be generated.

Clk	**Y_0**	**Y_1**	**Y_2**	**Y_3**
	1	0	0	1
1	1	1	0	0
2	1	1	1	0
3	0	1	1	1
4	1	0	1	1
5	0	1	0	1
6	0	0	1	0
7	1	0	0	1

Table 4. Pseudorandom pattern generated by LFSR in Figure 9

If we need we can modify the design in Figure 10 in order to allow the inclusion of the all-zeros pattern. It can be done by adding a NOR gate as is shown in Figure 10.

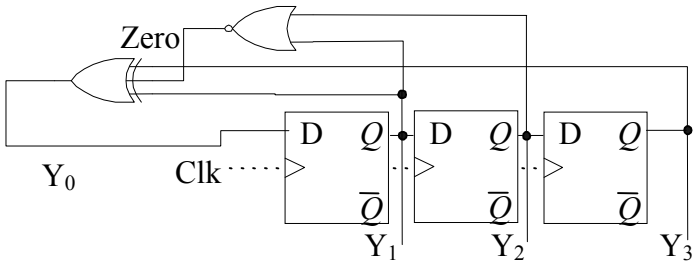


Figure 10. LFSR of Figure 10 with adjustment for all-0's pattern

The results are shown in Table 5, where eight distinct patterns are listed.

Clk	**Y_0**	**Zero**	**Y_1**	**Y_2**	**Y_3**
	0	1	0	0	1
1	1	1	0	0	0
2	1	0	1	0	0
3	1	0	1	1	0
4	0	0	1	1	1
5	1	0	0	1	1
6	0	0	1	0	1
7	0	0	0	1	0
8	0	1	0	0	1

Table 5. Pseudorandom pattern generated by LFSR in Figure 10

3 FAULT DIAGNOSIS

Introduction. A unit under test (UUT) fails when its observed behavior is different from its expected behavior. In case UUT is to be repaired, the cause of the observed error(s) must be diagnosed. Diagnosis consists of locating the physical fault(s) in a structural model of the UUT. The degree of accuracy to which faults can be located is called *diagnostic resolution*. No external testing can distinguish among *functionally equivalent faults* (FEF). The partition of all faults into distinct subsets of FEF defines the *maximal fault resolution*. A test that achieves the maximal fault resolution is said to be a *complete fault-location test*.

Repairing the UUT often consists of substituting one of its *replaceable units* (RU) referred as a *faulty* RU, by a good unit. Hence usually we are interested only in locating a faulty RU, rather than in an accurate identification of the real fault inside an RU. This diagnosis process is characterized by *RU resolution*. Suppose that the results of the test do not allow distinguishing between two suspected RUs *U1* and *U2*. We could replace now one of these RUs, say *U1* with a good RU and then return to the test experiment. If the new results are correct, the faulty RU was the replaced one; otherwise, it is the remaining one *U2*. This type of approach is an example of a *sequential diagnosis procedure*.

It is true to say that the diagnosis process is often *hierarchical*. There are two main types of the hierarchical diagnosis process and they are called: the *top-down diagnosis process* and the *bottom-up diagnosis process*.

In the *top-down diagnosis process* (system - boards - ICs) first-level diagnosis may deal with large RUs, such as boards containing many components, these are referred to as

field-replaceable units. The faulty board is then tested in a maintenance center to locate the faulty component on the board. Accurate location of faults inside a faulty IC may be also useful for improving its manufacturing process.

In the *bottom-up diagnosis process* (ICs - boards - system) a higher level is assembled only from components already tested at a lower level. This is done to minimize the cost of diagnosis and repair, which increases significantly with the level at which the faults are detected.

3.1 COMBINATIONAL FAULT DIAGNOSIS METHODS

This approach does most of the work before the testing experiment. It uses fault simulation to determine the possible responses to a given test in the presence of faults. The database constructed in this step is called a *fault table* or a *fault dictionary*. To locate faults, one tries to match the actual results of test experiments with one of the precomputed expected results stored in the database. The result of the test experiment represents a *combination* of effects of the fault to each test pattern. That's why this approach is called *combinational fault diagnosis method*.

3.1.1 Fault table. The fault table associates with each fault a set of test patterns that uniquely identifies the fault. For example, consider the information in Table 6, where an entry of $a_{jk} = 1$ indicates that fault k is detected by pattern j, if $a_{jk} = 0$ then fault k is not detected by pattern j. Now, if we know that a circuit passes all test patterns except the fourth pattern, we can deduce that the failure was due to fault *F8*. If, however, the circuit fails both test patterns *T3* and *T4*, the cause is either *F5* or *F8*.

Pattern	**Fault**							
	F1	F2	F3	F4	F5	F6	F7	F8
T1	1	0	1	0	0	1	0	0
T2	0	1	1	1	0	0	0	0
T3	0	0	1	0	1	0	1	0
T4	0	0	1	0	0	0	1	1

Table 6. Fault table

Another good example is shown in next Figure 11.

Pattern	Fault							Experiment		
	F1	F2	F3	F4	F5	F6	F7	E1	E2	E3
T1	0	1	1	0	0	0	0	0	0	1
T2	1	0	0	1	0	0	0	0	1	0
T3	1	1	0	1	0	1	0	0	1	0
T4	0	1	0	0	1	0	0	1	0	1
T5	0	0	1	0	1	1	0	1	0	1
T6	0	0	1	0	0	1	1	0	0	0

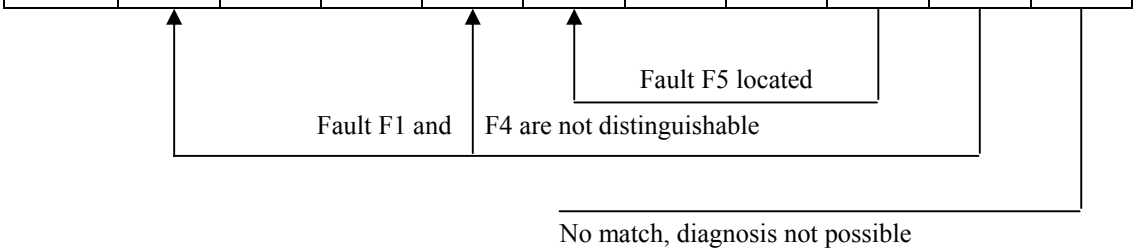


Figure 11. Fault table and the diagnosis

In this example the results of three test experiments *E1*, *E2*, *E3* are demonstrated. *E1* corresponds to the first case where a single fault is located, *E2* corresponds to the second case where a subset of two not distinguishable faults are located, and *E3* corresponds to the third case where no fault can be located because of the mismatch of *E3* with the column vectors in the fault table.

3.1.2 Fault dictionary. Fault dictionaries contain the *same* data as the fault tables with the difference that the data is *reorganized*. In fault dictionaries a mapping between the potential results of test experiments and the faults is represented in a more compressed and ordered form. For example, the column bit vectors can be represented by ordered decimal codes.

3.2 SEQUENTIAL FAULT DIAGNOSIS METHODS

In sequential fault diagnosis the process of fault location is carried out step-by-step, where each step depends on the result of the diagnostic experiment at the previous step. Such a test experiment is called *adaptive testing*. Sequential experiments can be carried

out either by observing only output responses of the UUT or by pinpointing by a special probe also internal control points of the UUT (*guided-probe testing*). In this chapter we will dedicate the separate subchapter to the guided-probe testing. Sequential diagnosis procedure can be graphically represented as *diagnostic tree*.

3.2.1 Fault location by edge-pin testing. In fault diagnosis test patterns are applied to the UUT step-by-step. In each step, only output signals at edge-pins of the UUT are observed and their values are compared to the expected ones. The next test pattern to be applied in adaptive testing depends on the result of the previous step. The diagnostic tree of this process consists of the fault nodes (rectangles) and test nodes (circles). A fault node is labeled by a set of not yet distinguished faults. The starting fault node is labeled by the set of all faults. To each fault node *k* a test node is linked labeled by a test pattern *Tk* to be applied as the next. Every test pattern distinguishes between the faults it detects and the ones it does not. The task of the test pattern *Tk* is to divide the faults in fault node *k* into two groups - detected and not detected by *Tk* faults. Each test node has two outgoing edges corresponding to the results of the experiment of this test pattern. The results are indicated as *passed* (P) or *failed* (F). The *diagnostic tree* in the Figure 12 below corresponds to the example considered in Figure 11. We can see that most of the faults are uniquely identified; two faults *F1, F4* remain indistinguishable. Not all test patterns used in the fault table are needed. Different faults need for identifying test sequences with different lengths. The shortest test contains two patterns the longest four patterns.

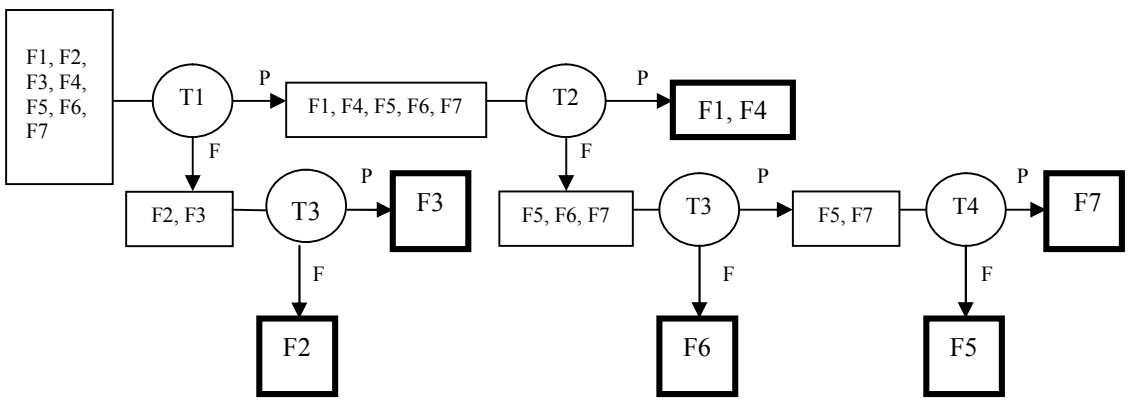


Figure 12. An example of a diagnostic tree

Rather than applying the entire test sequence in a fixed order as in combinational fault diagnosis, adaptive testing determines the next vector to be applied based on the results obtained by the preceding vectors. In our example, if *T1* fails, the possible faults are {*F2,F3*}. At this point applying *T2* would be wasteful, because *T2* does not distinguish among these faults.

3.2.2 Guided-probe testing. Guided-probe testing extends edge-pin testing process by monitoring internal signals in the UUT via a *probe*, which is moved (usually by an operator) following the guidance provided by the test equipment. The principle of guided-probe testing is to backtrace an error from the primary output where it has been observed during edge-pin testing to its source (physical fault) in the UUT.

Typical faults located by guided-probe testing are opens and defective components. An open between two points *A* and *B* is identified by a mismatch between the error observed at *B* and the correct value measured at the signal source *A*. A faulty device is identified by detecting an error at one of its outputs, while only expected values are observed at its inputs.

Unlike the fault-dictionary method, guided-probe testing is not limited to the SSF model. The concept of "defective component" covers any type of internal device fault detected by the applied test; this generality makes guided-probe testing independent of a particular fault model.

The most time-consuming part of guided-probe testing is moving the probe. To speed-up the fault location process, we need to reduce the number of probed lines. A lot of methods to minimize the number of probes are available.

Here are some *speed-up techniques* to achieve this goal with 3 explaining examples:

- Skip probing the output of a suspected device and directly probe its inputs. For the example in Figure 13, after observing an error at *i*, next step is to probe directly the inputs of *m*. The output *j* would be probed only if no errors are detected at the inputs, to distinguish between the faulty device *m* and an open between *j* and *i*.

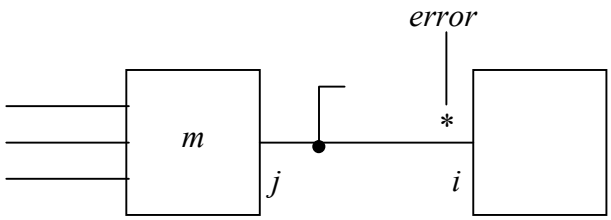


Figure 13. Case 1 when probing

- Probe only those inputs that can affect the output with errors (see Figure 14 below, the bold lines are affecting the erroneous output).

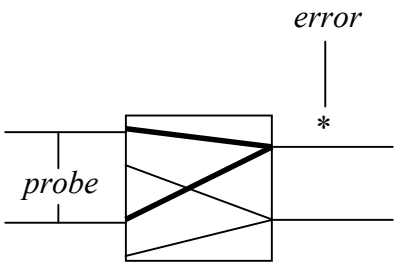


Figure 14. Case 2 when probing

- Among the inputs that can influence the output with errors, probe first the control lines; if no errors are detected at the control lines, then probe only those data inputs enabled by the values of the control lines. For the MUX shown in Figure 15, if no errors are found at the select inputs S_0 and S_1, the next line probed is D_3, because it is the input selected by $(S_0 , S_1) = 11$.

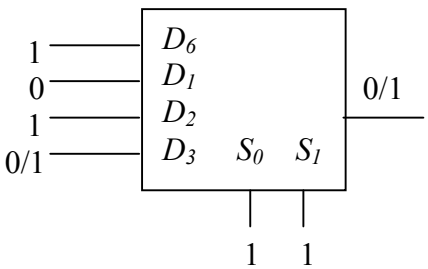


Figure 15. Case 3 when probing

- Use a fault dictionary to provide a starting point for probing.

Next will be discussed an example that is illustrated in Figure 16. Guided-probe testing technique is also used here. Our main goal is to find a faulty line in a given circuit. To achieve this goal, we can construct a diagnostic tree.

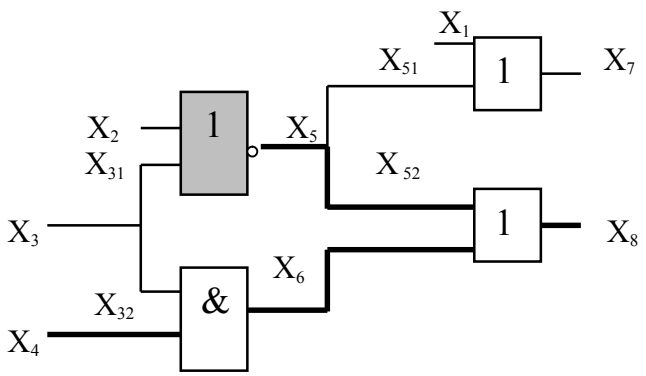


Figure 16. Given circuit

Let have a test pattern *1010* applied to the inputs of the circuit. The diagnostic tree created for this particular test pattern is shown in Figure 17. On the output x_8, instead of the expected value *0*, an erroneous signal *1* is detected. By backtracing (indicated by bold arrows in the diagnostic tree) the faulty component NOR - x_5 is located.

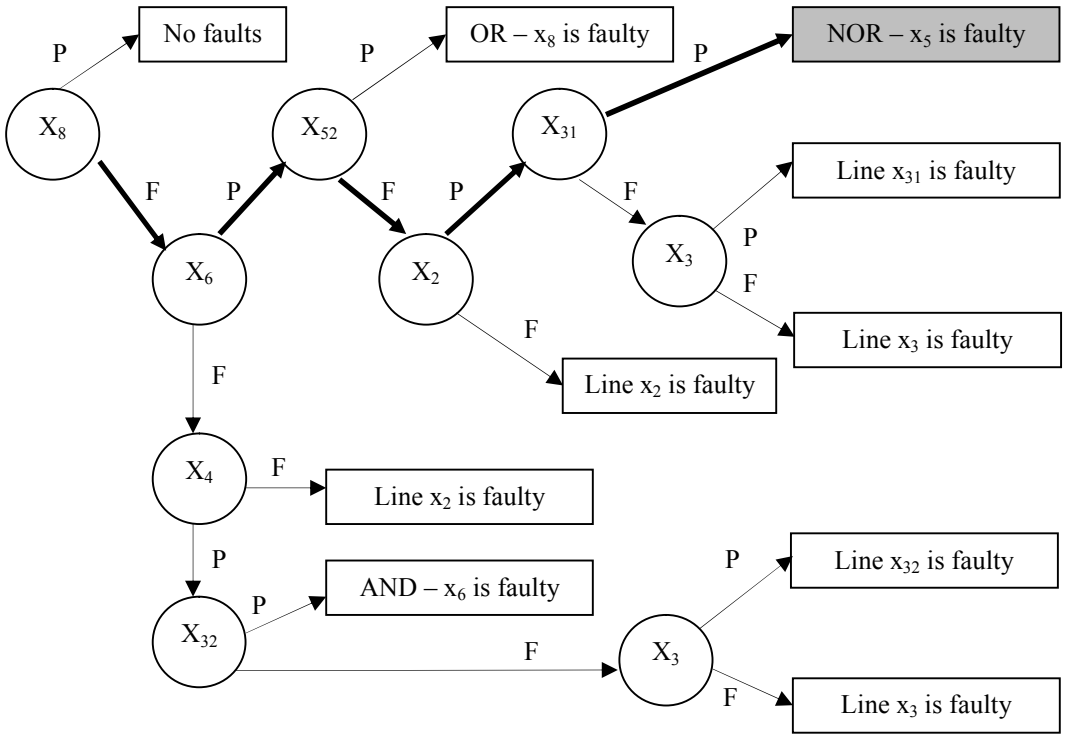


Figure 17. The diagnostic tree for the circuit in Figure 16

In conclusion must be said that diagnostic tree allows carrying out optimization of the fault location procedure, for example to generate a procedure with minimum average number of probes.

4 BOUNDARY SCAN

Introduction. Boundary-scan test (BST) is a method for testing boards. The BST standard interface was designed to test boards, but it is also useful to test ASICs. The BST interface provides a standard means of communicating with test circuits on-board an ASIC. We do need to include extra circuits on an ASIC in order to use BST. This is an example of increasing the cost and complexity (as well as potentially reducing the performance) of an ASIC to reduce the cost of testing the ASIC and the system.

The general boundary scan architecture is shown in Figure 18.

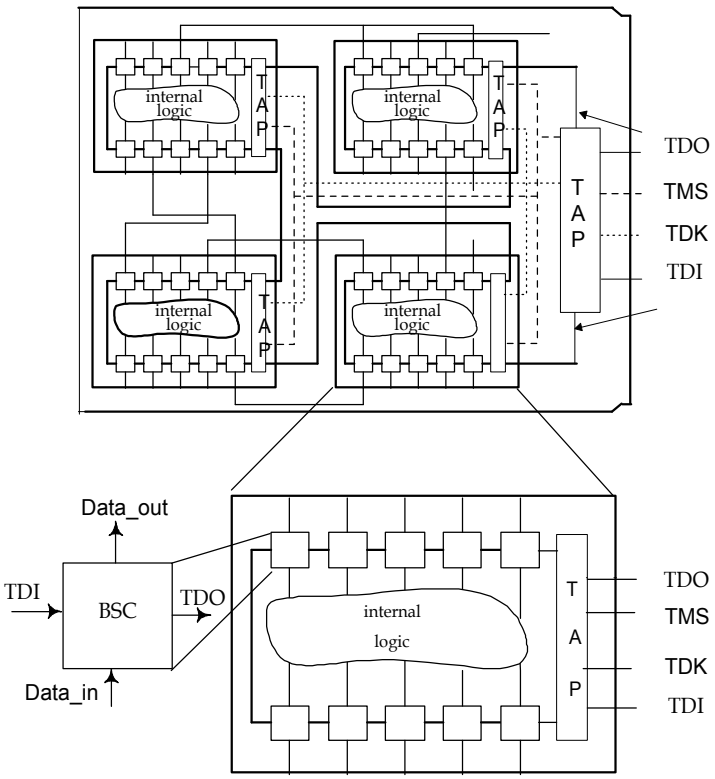


Figure 18. Boundary scan architecture

This configuration requires that the board and each IC that is part of the boundary scan include the following principal hardware components:

- A test access port (TAP) with four to five pins
- A group of registers: an instruction register (IR) and data registers (DRs)
- A TAP controller, a 16-state finite state machine

The boundary scan architecture allows configuring the cells for the following testing modes:

- External testing: interconnects between the chips
- Internal testing: testing of the logic within the chip

4.1 TEST ACCESS PORT

TAP controller uses various types of signals. They are the signals applied to the four mandatory pins and the optional test reset (TRST). The latter pin can reset the test logic asynchronously. The four mandatory pins include two data pins, test data input (TDI) and test data output (TDO), and two control pins, test mode select (TMS) and test clock (TCK).

Next is written the functions of all signals of the TAP controller:

- *TDI*. This signal allows the introduction of test data. The TDI of the board is connected to its counterpart on the first chip in the scan chain. This signal is shifted in the registers at the positive edge of the TCK and, when not in use, is kept high.
- *TDO*. Test data output allows scanning out of the test data. The TDO of the board is connected to its counterpart on the lat chip in the scan chain. Data are shifted out at the negative edge of TCK.
- *TCK*. The test clock operates the testing function synchronously and independent of the system clock. It controls the transfer to data and instructions among the TAP registers and shifting the data within any of the registers.

- *TMS*. The input stream to this pin is interpreted by the TAP controller and used to manage the various test operations.
- *TRST*. This is an optional signal whose purpose is to reset all testing logic asynchronously and independent of TCK.

4.2 REGISTERS

There are three mandatory registers: the instruction register, the bypass register, and the boundary scan register (BSR), which consists of collections of the boundary scan cells (BSCs). A brief description of these registers is given below in this subchapter. But first of all it is necessary to know about the functionality of the BSC.

Boundary scan cell. Common boundary scan cell may be used at the input or output pins. During normal operation the input signal is applied to the data-in pin and passes to the internal logic through MUX2. Thus the value of Test/Normal should be *0*, while the Shift-Load mode may be either *0* or *1*. When the same cell is used as an output pin, the data-in are from the internal logic of the chip and pass, through MUX2, to the output of the chip. An example of a boundary scan cell is shown in Figure 19.

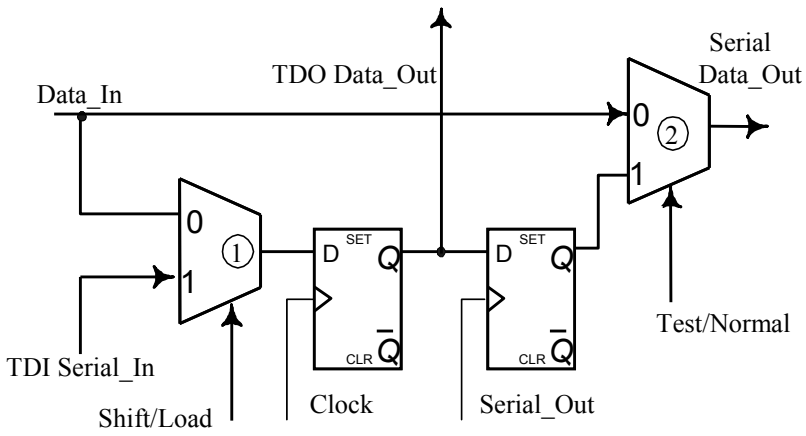


Figure 19. Basic boundary scan cell

For the test mode, the data are coming through TDI, thus the Shift/Load mode should be *1*. The data are latched for internal testing or to be shifted to the next BSC when the clock is activated.

Bypass register. The bypass register is set to logic *0* at the rising edge of TCK when the TAP controller is in the Capture-DR state. Use of the bypass register allows the signal at the TDI to pass directly to the TDO of the chip, thus bypassing all the other BSCs of the chip. Figure 20 shows the flow of data through the bypass register.

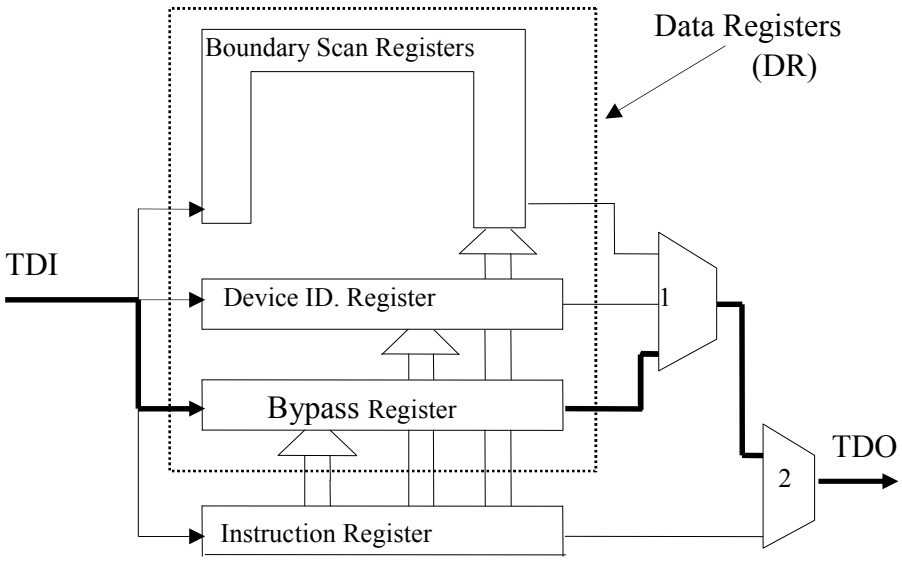


Figure 20. Various registers and flow of data through the bypass register

Boundary scan register. The boundary scan register consists of all the BSC cells on the periphery of the chip. These registers are also shown in the previous Figure 20. Boundary scan register is part of the testing of the interconnects and of any logic between the boundary scan ICs on the board.

Instruction register. The instruction register is a serial-in, parallel-out register. All the instruction registers of the IC are connected in a chain. In this register the appropriate instructions are shifted in serially and the individual instructions are captured in parallel.

Device identification register. The device identification register is optional, but if included on the IC, it should comply with the standard. It must be a 32-bit-long parallel-in and serial-out. It is intended to contain the manufacturer's number and the version number. This information facilitates verifying that the correct IC is mounted in a correct position and it is the correct version of the chip. Unlike the other registers, the information is not passed to an input latch.

4.3 TAP CONTROLLER

The TAP controller has three main functions:

1. Loading the instructions in the IR.
2. Providing control signal to load and shift the test data into TDI and out of TDO.
3. Performing some test actions, such as capture, shift, and update test data.

The state diagram of the TAP controller is shown in Figure 21. Some of the states correspond to actual operations on the data (DR) or the instructions (IR), while others allow some flexibility in the flow of operations.

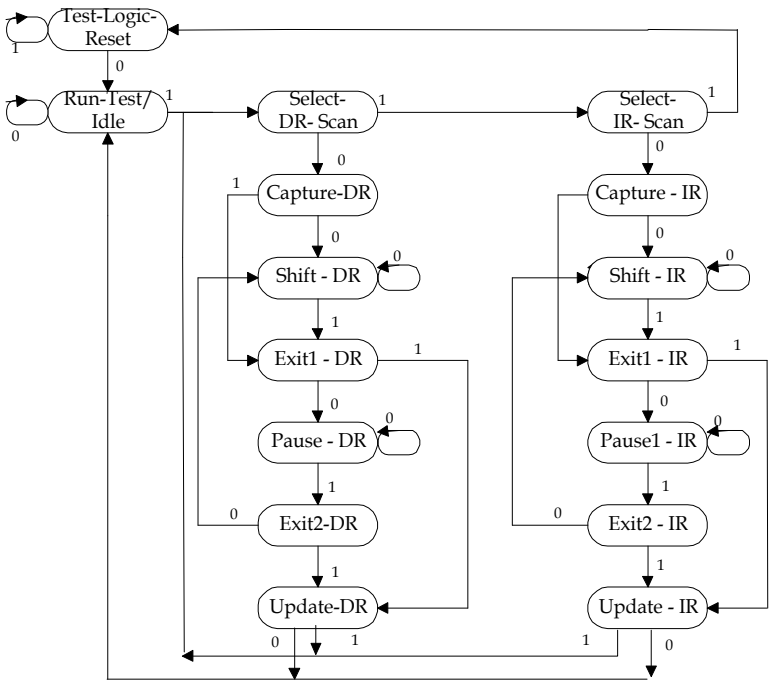


Figure 21. Test access port controller

4.3.1 Controller's states. At power on, the controller is in the Test-Logic-Reset state. It remains as long as TMS is high and the circuit is in normal operation mode. As soon as TMS changes to logic 0, the controller is in the Run-Test-Idle state.

To start testing, the instruction needs to be loaded into IR. For this, TMS is held high and the TCK is clocked twice for the controller to reach the Select-IR-Scan state. Now

TDI and TDO are connected to IR, and all IR registers on the board are serially connected. Next, the controller passes to the Capture-IR state with TMS = 0. Once the instruction is loaded in the IRs, then, with TMS still low, the controller stays in the Shift-IR state for as many clock cycles as needed by the test mode. In this state, the previously captured data are shifted via TDI and via TDO. If shifting is not needed, TMS = 1 and the controller bypasses Shift-IR and enters Exit1-IR state. The latter state as well as any of the other exit states is temporary. At the next positive edge of the clock, there is transition to another state. If TMS = 0, the next state is Pause-IR, and the control remains in this state until TMS = 1. The Pause-IR state is needed when the shift is done in a chain of different lengths. From this state, the control goes to Exit2-IR, then to Shift-IR if TMS = 0, or to Update-IR, if TMS = 1. The controller enters this state once the shifting process has been completed. The new data are latched into their parallel outputs of the selected data registers at the falling edge of the TCK. Depending on the value of TMS, the next state is either Run-Test-Idle or Select-DR-Scan. When the controller is in the DR branch of the state diagram, it performs on the IR operations similar to those described above.

4.3.2 Instruction set. The controller utilizes only a few instructions. Only three of these are mandatory: BYPASS, EXTEST, and SAMPLE/PRELOAD. The most commonly used optional instructions are IDCODE, INTEST, and RUNBIST. Below you can see the functions of three mandatory as well as most commonly used optional instructions.

BYPASS Instruction. This mandatory instruction permits bypassing of the current IC. It places the one-bit bypass register between TDI and TDO of the chip when another IC is being tested.

EXTEST Instruction. This is mandatory instruction, which is actually the primary reason for boundary scan. It allows testing of the connectivities of the various pins of the ICs mounted on the board. The fault models used are stuck-at, bridging faults, and opens. For present technology circuits, noise faults should also be considered (for example, such noise failure as ground bounce).

SAMPLE/PRELOAD Instruction. This instruction is mandatory and is used to scan BSR without interrupting the normal operation of the internal logic. It supports two

functions: sampling the normal operations of the chip and preloading data for another test operation into the latched parallel outputs of the BSCs.

IDCODE Instruction. Although this optional instruction does not involve testing of the board, it helps identifying misplaced ICs. Often, it is difficult to distinguish between similar devices, and discovering the reason for malfunction of the board may take unnecessarily a long time.

INTEST Instruction. This is another optional instruction. It allows static testing of a particular IC using a bed-of-nails fixture and pin probing of ATE. The test patterns are applied to the input of the chip and the response captured at the outputs. The test data are applied one at a time at the rate of TCK.

RUNBIST Instruction. This instruction is also optional. It causes the execution of BIST test provided on the IC selected. It requires minimum data from outside the chip since in BIST the test patterns are generated internally on the chip. These patterns are applied dynamically. The instruction is run when the TAP controller is in its Run-Test-Idle state. The result of the test is captured when the controller is in Capture-DR state.

CLAMP Instruction. This optional instruction is used to control the output signal of a component to a constant level by means of a BSC. This is useful to hold values on some pins of the circuit, which are not involved in the test. These required signals are then loaded with other test patterns every time they are needed. This instruction, although useful, increases test application time.

HIGHZ Instruction. The HIGHZ instruction is optional and forces all outputs of a component to a high-impedance state. It is used, for example, when an in-circuit test is required for testing a non-BS compliant component.

4.4 BSDL

Since 1990 when the IEEE 1149.1 (Test Access Port and Boundary Scan Architecture) standard was approved, implementation of the standard has accelerated. As more people became aware of and used the standard, the need for a standard method for describing

IEEE 1149.1-compatible devices was recognized. The IEEE 1149.1 working group established a subcommittee to develop a device description language to address this need.

The subcommittee has since developed and approved an industry standard language called Boundary Scan Description Language (BSDL). BSDL is a subset of VHDL (VHSIC Hardware Description Language) that describes how IEEE 1149.1 is implemented in a device and how it operates. BSDL captures the essential features of any IEEE 1149.1 implementation.

Now let's take a look at the elements of BSDL. A BSDL description for a device consists of the following elements:

- Entity descriptions
- Generic parameter
- Logical port description
- Use statements
- Pin mapping(s)
- Scan port identification
- Instruction Register description
- Register access description
- Boundary Register description

Entity descriptions – The entity statement names the entity, such as the device name (e.g. SN74ABT8245). An entity description begins with an entity statement and terminates with an end statement.

```
entity XYZ is
    {statements to describe the entity go here}
end XYZ
```

Generic parameter – A generic parameter is a parameter that may come from outside the entity, or it may be defaulted, such as a package type (e.g. "DW").

```
generic (PHYSICAL_PIN_MAP : string : = "DW");
```

Logical port description – The port description gives logical names to the I/O pins (system and TAP pins), and denotes their nature such as input, output, bidirectional, and so on.

```
port (OE: in bit;
        Y: out bit_vector(1 to 3);
        A: in bit_vector(1 to 3);
      GDN, VCC, NC: linkage bit;
      TDO: out bit;
      TMS, TDI, TCK: in bit);
```

Use statements – The use statement refers to external definitions found in packages and package bodies.

```
use STD_1149_1_1994.all;
```

Pin mapping(s) – The pin mapping provides a mapping of logical signals onto the physical pins of a particular device package.

```
attribute PIN_MAP of XYZ : entity is
PHYSICAL_PIN_MAP;
constant DW: PIN_MAP_STRING: =
  "OE: 1, Y: (2,3,4), A: (5,6,7), GND:8, VCC:9, "&
        "TDO:10, TDI:11, TMS:12, TCK:13, NC:14";
```

Scan port identification – The scan port identification statements define the device's TAP.

```
attribute TAP_SCAN_IN of TDI : signal is TRUE;
attribute TAP_SCAN_OUT of TDO : signal is TRUE;
attribute TAP_SCAN_MODE of TMS : signal is TRUE;
attribute TAP_SCAN_CLOCK of TCK : signal is (50.0e6, BOTH);
```

Instruction Register description – The Instruction Register description identifies the device-dependent characteristics of the Instruction Register.

```
attribute INSTRUCTION_LENGTH of XYZ : entity is 2;
attribute INSTRUCTION_OPCODE of XYZ : entity is
  "BYPASS (11), "&
  "EXTEST (00), "&
  "SAMPLE (10) ";
attribute INSTRUCTION_CAPTURE of XYZ : entity is
        "01";
```

Register access description – The register access defines which register is placed between TDI and TDO for each instruction.

```
attribute REGISTER_ACCESS of XYZ : entity is
  "BOUNDARY (EXTEST, SAMPLE), "&
  "BYPASS (BYPASS) ";
```

Boundary Register description – The Boundary Register description contains a list of boundary scan cells, along with information regarding the cell type and associated control.

```
attribute BOUNDARY_LENGTH of XYZ : entity is 7;
attribute BOUNDARY_REGISTER of XYZ : entity is
  "0 (BC_1, Y(1), output3, X, 6, 0, Z), "&
  "1 (BC_1, Y(2), output3, X, 6, 0, Z), "&
  "2 (BC_1, Y(3), output3, X, 6, 0, Z), "&
  "3 (BC_1, A(1), input, X), "&
  "4 (BC_1, A(2), input, X), "&
  "5 (BC_1, A(3), input, X), "&
  "6 (BC_1, OE, input, X), "&
  "6 (BC_1, *, control, 0)";
```

REFERENCES

[1] MOURAD, S., ZORIAN, Y. *Principles of Testing Electronic Systems*. John Wiley & Sons, Inc., New York, 2000.

[2] ABRAMOVICI, M., BREUER, M.A., FRIEDMAN, A. *Digital Systems Testing and Testable Design*. IEEE Press / AT&T, New York, 1990.

[3] Michael John Sebastian Smith. *Application-Specific Integrated Circuit*. Addison Wesley Professional, 1997.

[4] *Boundary-Scan Tutorial*. ASSET InterTech, Inc., 1998.

[5] Supporting theoretical notes on diagnostics URL: http://www.pld.ttu.ee/diagnostika/theory/faultdiagnosis.html