Chapter 3

TEST GENERATION TECHNIQUES AND ALGORITHMS

Raimund Ubar [(1)], Elena Gramatová[(2)], Mária Fischerová[(2)]
[(1)]Tallinn University of Technology, Tallinn, Estonia
[(2)]Institute of Informatics of the Slovak Academy of Sciences, Bratislava, Slovakia

Abstract: This chapter describes different approaches to test generation, fault simulation and fault diagnosis in digital circuits and systems. First, an overview about the main techniques of logic level structural test generation for digital circuits is given. Then these techniques are explained in terms of graph technique based on structurally synthesised BDDs. Differently from the classical BDD-based test generation, the emphasis is given here to the topological aspects of test generation on SSBDDs in terms of path activation tasks on graphs, similarly to path activation in logic level circuits. Such a topological view on SSBDDs allows easily to generalise logic level test generation algorithms for higher level test generation purposes. In more details, register transfer level test generation and instruction set level test generation for microprocessors are considered. Several methods of gate-level fault simulation like parallel, deductive and critical path tracing methods are discussed. Afterwards macro-level fault simulation for logic circuits and hierarchical fault simulation for digital systems are considered. Advantages and disadvantages of all these methods are highlighted. In the subchapter on fault diagnosis, combinational and sequential fault localisation procedures are described. Finally, test generation methods for RAM are discussed. An overview about traditional methods like March, Walkpat, Galpat a.o. is given. In more detail the capability of March test to detect different faults like stuck-at-faults, transition, addressing and coupling faults is analysed. Finally, the problems of testing pattern sensitivity faults and layout related faults is discussed.

Keywords: faults, test pattern generation, fault simulation, logical, functional, defect-oriented, hierarchical test generation techniques, memory faults, March algorithms, BDD, fault diagnosis

3.1 LOGIC-LEVEL TEST GENERATION

Test generation itself plays a key role in various processes such as logic optimisation, verification, design for testability, and built-in self-testing where the efficiency of the combinational test pattern generation algorithms is an important issue [1], [2], [3]. Frequently, the circuit designer will provide a limited subset of the functional test patterns for a **device under test** (DUT), which typically cover only 70 to 75 % of the total number of faults. Testing for only 75 % of the modelled defects is not sufficient. Thus, the importance of **automatic test pattern generation** (ATPG) algorithms at the structural level is undisputed. ATPG is the application of algorithmic based software to generate vectors – test patterns. The traditional goal of ATPG algorithms is to achieve high fault coverage by producing a small volume of test patterns. Therefore the first step after processing of the design description involves establishing the fault model to be used, and the faults have to be enumerated. **Fault models** represent defects and a fault list is defined for DUT. Obviously, inputs to ATPG systems consists are a DUT netlist (often based on standard cell libraries), a list of targeted faults and requirements to test vectors quality [1].

Historically, the single **stuck-at fault model** (**SAF model**; *stuck_at 0* – SAF0, *stuck_at 1* – SAF1) has been widely accepted as a standard fault model for the test pattern generation algorithms. The usage of the SAF model will continue as long as ATPG exists. Although the SAF model cannot guarantee the highest quality of defect testing, especially for CMOS integrated circuits [4], [5], [6], it is still the key fault model of the structural TPG algorithms. Its importance is due to its simplicity, tractability, logical behaviour, measurability and adaptability [6]. Various TPG algorithms targeted to other fault models (e.g. bridging faults, delay faults) or other testing types (I_{DDQ} and at speed testing) are often based on the structural TPG strategies using the SAF model. Each test pattern generation algorithm is obviously evaluated by the following measures:

- **test effectiveness** = (detected and proven non testable faults)/total faults
- **fault coverage** = detected faults/total faults
- test generation time
- length of the generated test set (test volume).

3.1.1 Structural test generation algorithms

The structural TPG algorithms can be applied for multiple purposes because they can [1]:

- generate test patterns

- find redundant or unnecessary logic
- verify whether one circuit implementation matches another circuit implementation.

Since the scan-based design techniques are increasingly used for complex circuit designs, the structural test generation for combinational circuits becomes more important. The test generation problem can be viewed as a finite space search problem of finding appropriate logic assignments to the primary inputs such that the given fault is detected. The size of the search space is exponential in the number of primary inputs and the test generation problem is proven to be **NP-complete problem**. It means that no polynomial expression for the computing time function was found, and the problem is presumed to have exponential complexity. Therefore it is very important to develop efficient techniques to speed up the test generation process producing an optimal test set volume.

Besides of simple test generation algorithms: **exhaustive** (every possible test is applied to the *n* input ports of DUT – it means 2^n logic values), **pseudo-exhaustive** (some portion of all possible 2^n logic values), **random**, **pseudo-random** and **deterministic TPG** algorithms remain still of great interests in the research field. The pseudo-random TPG means that a circuit is divided into cones (a cone is a part of DUT with one primary output and those primary inputs, which are linked to this primary output) and a test set is generated randomly for each cone. If a random or pseudo-random TPG technique is used a **fault simulator** has to be applied for fault coverage computation for a defined fault list (see *Figure 3-1*).

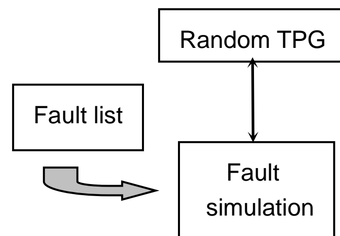


Figure 3-1. ATPG with a random TPG and a fault simulator

The random test pattern generation produces test patterns in the shortest time but they do not have to achieve sufficient high fault coverage. Thus the random TPG effectiveness is not too high for complex circuits in comparison with the deterministic TPG algorithms.

The fault coverage values are effective only at the beginning of the random TPG process. After exceeding a specific number of patterns a rise in the fault coverage is very slow. Usually the random TPG does not have to produce the highest fault coverage value (the standard expectation is 95 % -

99,9 % in the semiconductor industry [14]). It is caused by faults due to a **fault resistant** problem. It means that a special pattern inside DUT must be assigned.

An example of the fault resistant problem is given in *Figure 3-2* where only one out of 2^{32} (4 billion) patterns detects the SAF1 at the output of a circuit. Obviously 20 % - 40 % of faults are typically random pattern resistant.

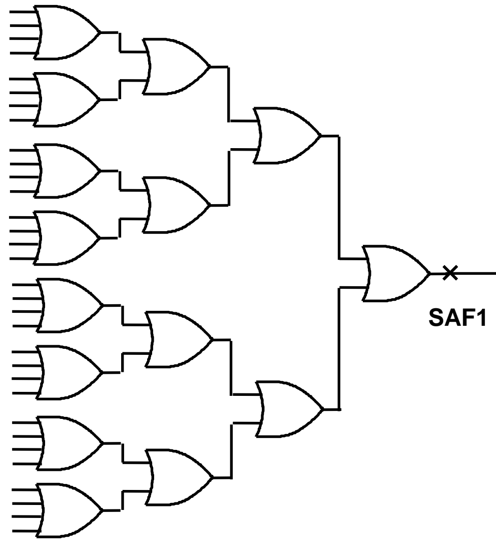


Figure 3-2. Fault resistant state (SAF1 at the output)

On the other hand, deterministic TPG algorithms assume that only a single fault is injected into the DUT structure for which a test is generated. Therefore any deterministic TPG run longer than the random TPG.

Beside this restriction, another problem in deterministic TPGs is the problem of finding right logical values for cells inside the DUT structure that cannot be assigned uniquely during the TPG procedures. Selection of the values is based on some specific rules or heuristics, and a value conflict can be announced during test generation for an injected fault. Each conflict has to be solved by returning back to a node where a new value assignment can be done – a decision node (point). This step is named **backtrack**. Then the deterministic algorithm speed depends also on the number of backtracks during the test generation process. Many algorithmic and heuristic strategies have been developed for decreasing the number of backtracks based on finding the conflict point in the DUT structure as quickly as possible.

The deteministic TPG should find a test pattern for a fault if it is detectable. Therefore it is important to use the deterministic TPG, e.g. for

hardly detectable faults. *Figure 3-3* shows in general the effectiveness of random and deterministic TPG algorithms.

The total number of backtracks can also be reduced by fault simulation applied to each new generated deterministic test vector to get he list of covered faults, and then test patterns are not generated by the deterministic TPG for the faults already covered.

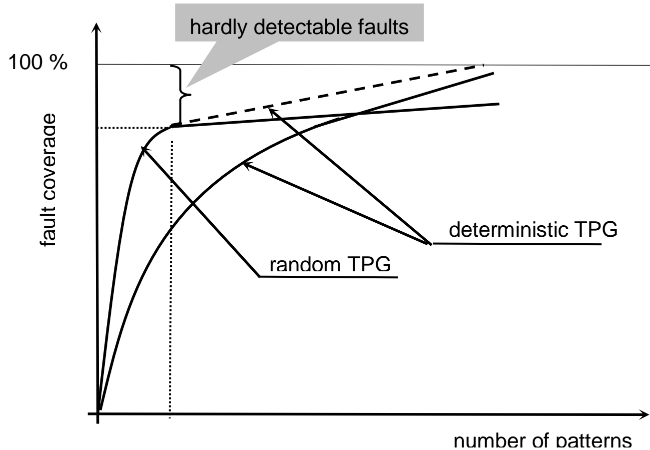


Figure 3-3. Effectiveness of the random and deterministic TPGs

A fault simulator must classify the given target faults in DUT as detected or undetected by a given test vector, then all faults covered by the vector are deleted from the fault list and the deterministic TPG is applied to other uncovered fault. It is known that the fault simulation speed is higher than the speed of any deterministic TPG algorithm because the process of finding covered faults is performed by tracing the DUT structure only twice (fault-free simulation, fault lists propagation) for one test vector. Thus, the typical and effective ATPG construction is shown in *Figure 3-4*. It means the random TPG is used at the first TPG phase running together with the fault simulator and the deterministic TPG phase is used only for hardly detectable faults also linked to the fault simulator.

Both the fault simulator and the deterministic TPG run over the same list of faults. The deterministic TPG algorithm is used for hardly detectable faults or after a limit of defined fault coverage is achieved by the random TPG [1]. Developed TPG algorithms are classified into 5 groups [1]:

1. TPG based on **path sensitisation** techniques
2. Simulation-based TPG methods
3. TPG using Boolean satisfiability (a Boolean expression or equation)
4. TPG based on implication graph methods
5. TPG methods based on genetic algorithms.

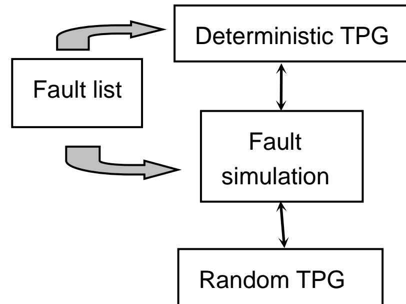


Figure 3-4. ATPGs with a fault simulator

Most of the existing professional and academic ATPG systems use the path sensitisation technique; this principle is based on faulty signal sensitisation through each cell and on the **D-calculus** defined by Roth [8]. *Figure 3-5* presents the path sensitisation through basic gate NOR for fault SAF0 propagated from its input to output (logical 1 is forced to the faulty input).

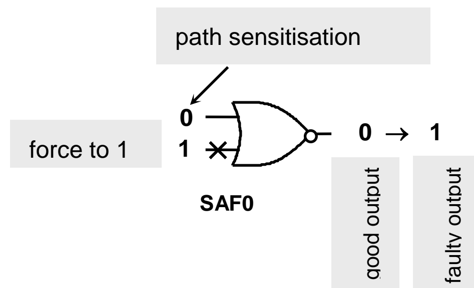


Figure 3-5. Path sensitisation through basic cell NOR

The path sensitisation rules for basic gates – AND, NAND, OR, NOR, XOR, XNOR were defined (see *Figure 3-6*). The sensitive path is always created through cells NOT, BUFF, XOR and XNOR.

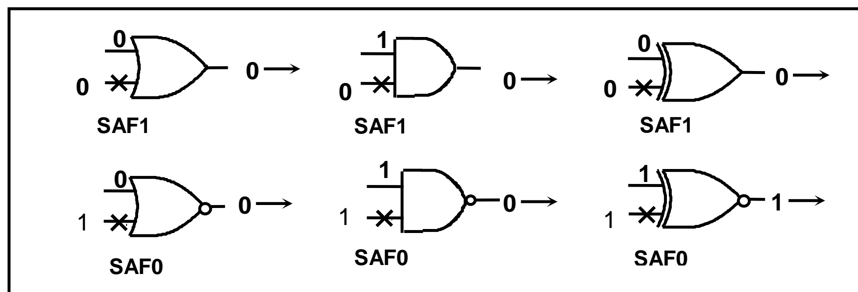


Figure 3-6. Path sensitisation through basic cell

The path sensitisation through complex cells has to be defined according to their structure; an example is shown in *Figure 3-7*. Three different input vectors (1000, 1010, 1001) can sensitise the fault on gate AND.

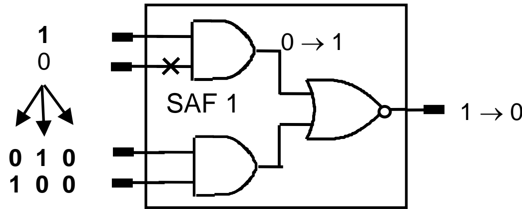


Figure 3-7. Path sensitisation through a complex cell

The 5-value logic model has been defined using the values 0, 1, x, D, ¬D; the value *x* means don't care value, and D, ¬D represent faulty signals SAF0 and SAF1, respectively. The value D represents logical value 1 in the fault free state and 0 in the faulty state. An example of the path sensitisation for SAF0 is shown on a circuit presented in *Figure 3-8*. The value D is propagated through the sensitive path using the following logic values: 0 on gate OR, 1 on gate AND, 0 or 1 on gate XOR. If the sensitive path is found from the fault site to a primary output, some values have to be assigned from the primary inputs (it means logical value 1 at the input of gate AND and logical value 0 at the input of gate XNOR inside the circuit in *Figure 3-8*).

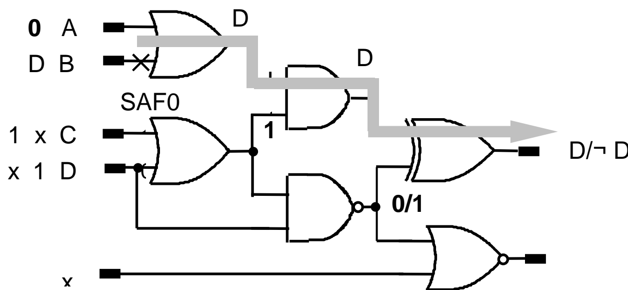


Figure 3-8. Path sensitisation using D calculus

Finding the right values at the primary inputs for confirmation of assigned values inside the circuit is realised by a **backtrace** through the circuit structure from the node of a non-confirmed value to a primary input. This step is named value (line) justification. The logical values 1 and 0 inside the circuit presented in *Figure 3-8* are justified by the pattern (CD) = (x1). The path sensitisation methods at the logical level of circuit representation are currently the most preferred ATPG methods and consist of the following 3 basic steps [1], [2], [9], [15]:

1. **Fault sensitisation** (fault activation, fault excitation), in which a SAF is activated by forcing the signal to an opposite value as the fault value (ensuring difference between good and faulty circuits).
2. **Fault propagation** (using path sensitisation), in which the fault effect is propagated through one or more paths to primary output(s) of the circuit. In general, the number of paths may rise exponentially with the number of logical gates in the circuit.
3. **Line justification**, in which the internal signal assignments previously used to sensitise a fault or propagate its effect is justified by setting the primary inputs of DUT; one example is shown in *Figure 3-9*, where two logical values 1 and 0 have to be justified. The logical value 1 on the primary inputs C and D justifies both desired values.

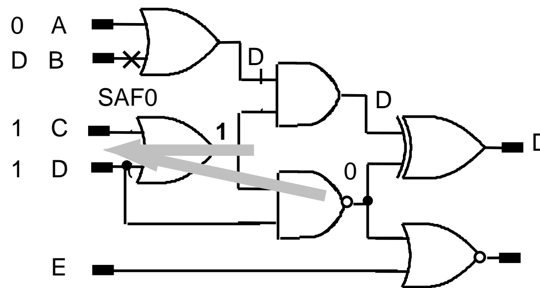


Figure 3-9. Line justification

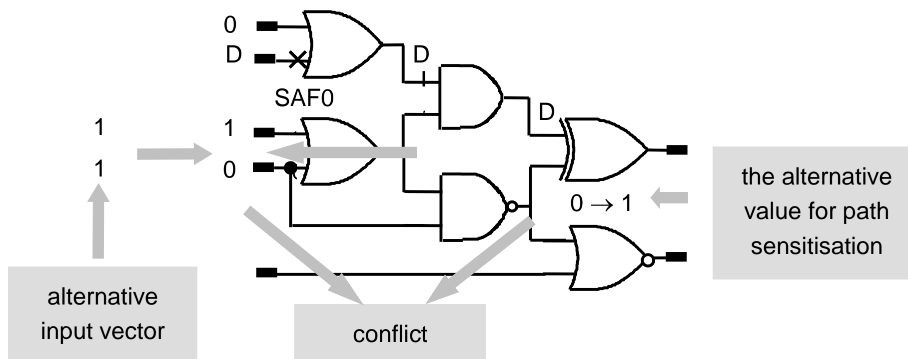


Figure 3-10. Conflict situation during the justification step

During the path sensitisation and the line justification some inconsistent states can arise and this has to be solved by alternative assignments – a backtrack must be used to a node in DUT where a new value assignment can be done. An example with two possibilities is shown in *Figure 3-10*. One alternative solution is a new assignment on the primary input and other one

is to return to the second fan-out and to change the value for path sensitisation through cell XOR (new logical value is 1).

Many algorithms based on the path sensitisation technique have been developed and implemented in various ATPG systems. All these algorithms use the mentioned 3 basic steps but their effectiveness depends on different heuristics and combinations of the mentioned steps. After the first TPG algorithm development – **D algorithm** [8] and its heuristics successors **PODEM** (1981; the branch and bound search algorithm) [7] and **FAN** (1981; fan-out-oriented algorithm) [9], many other techniques and heuristics have been developed improving and speeding up the existing TPG systems. They are e.g. SOCRATES (using static and dynamic learning procedures) [10], TOPS algorithm (based on the defined signal line as dominators) [11], recursive learning algorithm [13], EST (using defined evaluation frontier) [12] and some their modifications. In the FAN algorithm several progressive concepts were defined for decreasing the number of backtracks:

- The unique sensitisation procedure – values assignment to signals on gates involved in all sensitive paths for an investigated fault – this procedure is applied immediately after fault sensitisation.
- Application of the multiple backtracing procedure to primary inputs.
- Immediate implications (backward and forward) – values assignment for uniquely determined signals, places of backtracks – fan-out nodes of DUT [9]. The implication procedures are demonstrated in *Figure 3-11* using c17 ISCAS'85 benchmark circuit.

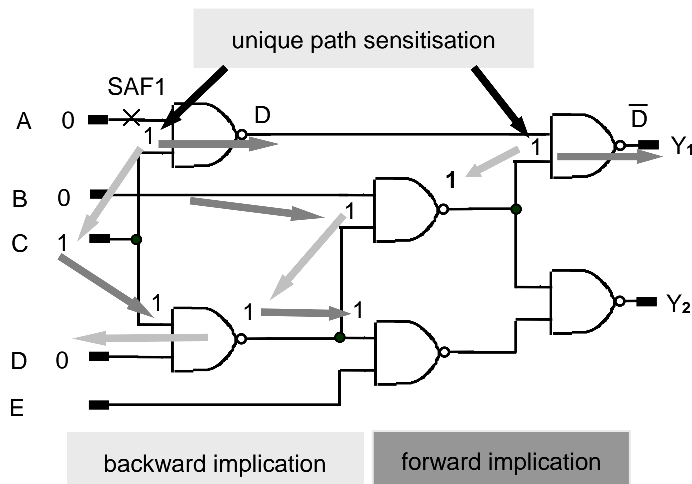


Figure 3-11. Forward and backward implications on c17 benchmark circuit

The research continues by improving the deterministic algorithms and new heuristics or algorithmic formulas have been published e.g. [16], [17],

[18], [19], [20]. *Table 3-1* shows the history of accelerating combinational ATPG algorithms and systems [1].

Table 3-1. TPG algorithms progress

Algorithm	Speed evaluation	year
D	1	1966
PODEM	7	1981
FAN	23	1983
TOPS	292	1987
SOCRATES	1574	1988
Waicukaiski	2189	1990
EST	8765	1991
TRAN	3005	1993
Recursive learning	485	1995
Tafertshofer	25057	1997

Nowadays, the following TPG algorithms have been published: SPIRIT [15], ATOM [17], STAR-ATPG [18] and some new techniques have been developed for speeding up the deterministic TPG process as dynamic decision ordering, conflict driven recursive learning and conflict learning [19]. The number of backtracks is the key step of the test generation speed.

The structural TPG algorithms can also be used for **defect-oriented testing** using an **implicit fault model**. The implicit fault model means that some patterns have to be set up on each cell of DUT during testing, e.g. for cell NOR (*Figure 3-12*) it is necessary to apply test patterns – (00, 01, 10). The same patterns, named also fault conditions, can be used for gate OR.

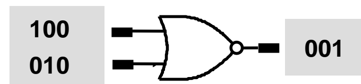


Figure 3-12. Implicit fault model

This fault model is suitable for defect-oriented testing [5] and I_{DDQ} testing [21]. In defect-oriented test pattern generation we need information about coverage of expected defects by means of specified vectors for each cell integrated in DUT. An example is demonstrated in *Table 3-2* for gate NOR (with two inputs A, B and output Q) where notation A/B means short between 2 nodes: A, B; "*" means its coverage by a corresponding vector.

Table 3-2. Fault table for NOR (Gn – ground node and Vd – power node)

vector	A/B	A/Q	A/Gn	A/Vd	B/Q	B/ Gn	B/Vd	Q/ Gn	Q/Vd
00		*		1	1		1	1	
01	1				1	1			1
10	1	1	1						1

Similar strategies based on the path sensitisation methods can be used for the test generation process using the implicit fault model. A selected pattern for a cell in DUT instead of faulty value D or ¬ D is propagated through sensitive paths and justified from primary inputs. An example is shown in *Figure 3-13*; the first pattern (AB) = (00) is selected for test generation. The received test vector (ABCDE) = (00011) covers 6 patterns on basic gates in the circuit.

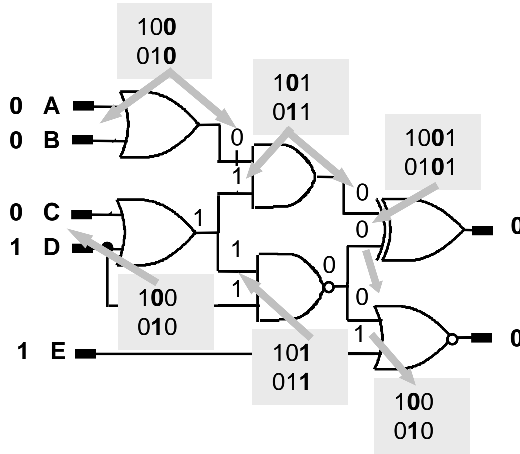


Figure 3-13. Test pattern generation using the implicit fault model

If some complex cells are used in a DUT structure, e.g. cell AN1 with function: Q = NOR (AND(A,B), AND(C,D)); fault conditions for AN1 could be defined with regard to stuck_at fault coverage of basic cells (ANDs, NOR) or results from defect analysis (described in Chapter 2). An example of fault conditions for cell AN1 is shown in *Figure 3-14*.

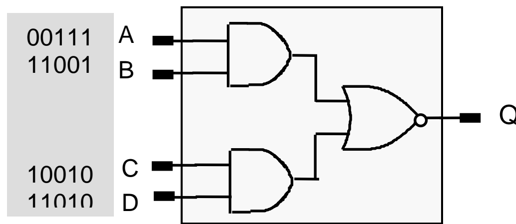


Figure 3-14. Implicit fault model – test patterns for complex cell AN1

Current-based testing (I_{DDQ} testing) is a completely different paradigm in comparison with the classical voltage-based testing (described in Chapter 7). The fault model cannot be restricted to the single stuck-at fault assumption

because the measured unit is the current at the chip-level. Test pattern generation techniques use the following fault models:

- The pseudo-stuck-at fault model (it means a fault is represented by SAF1 or SAF0 injected on a DUT node - but its manifestation has not to be propagated to a primary output).
- The implicit fault model based on the toggle test set (application logical 1 and logical 0 to all nodes in DUT).
- The implicit fault model based on specified test patterns = the fault conditions described above.

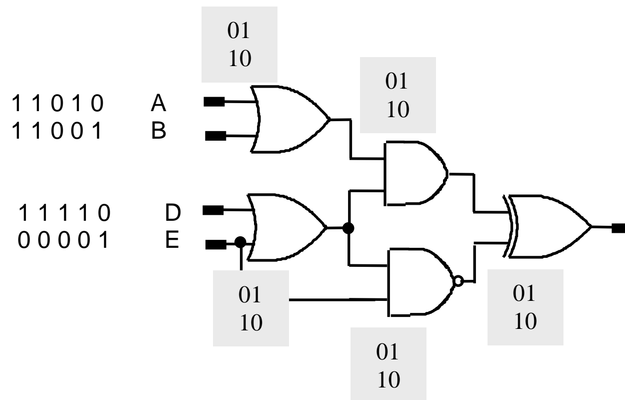


Figure 3-15. Test generation for I_{DDQ} testing using the toggle test set

Test pattern generation for I_{DDQ} testing is simpler than for classical voltage testing because the injected fault or a logical value has not to be observable on a primary output. It is enough to find a test vector that sets up defined values on gates inside DUT. An example for the test pattern generation using the toggle test for I_{DDQ} testing is shown in *Figure 3-15*. The test set of 5 patterns is generated for covering all desired values (toggle test patterns) on circuit cells.

Example 3-1: Consider test set generation for circuit c17 (*Figure 3-11*) by the random and deterministic TPG algorithms. There are 34 SAF0 and SAF1 in the c17 fault list. Their coverage is 100 %, e.g. by 8 random test patterns (ABCDE) = (00100, 10010, 00011, 00011, 10100, 11010, 11110, 10111) or by 5 deterministic test patterns (ABCDE) = (1x1010, 0110x, 01111, x00x1, 100x0), *x* is don't care value. If I_{DDQ} testing is used for some faults (not detected by classical voltage testing), the combined test patterns should be applied: e.g. 5 patterns for voltage and current measurements (ABCDE) = (100100, 01101, 11111, x10x0, x00x1) and 2 only for voltage testing (ABCDE) = (x101x, 001xx).

Almost all digital systems are realised as sequential circuits. These circuits contain combinational logic and flip-flops. Their testing is more complex than combinational circuit testing for two reasons [1]:

1. Internal memory states. The circuit has internal memory and its state is not known at the beginning of testing. A test must, therefore, initialise the circuit to a known state. After test inputs are applied, the final state of the internal memory must be inferred only indirectly from primary outputs. Only in special cases the internal memory can be made controllable and observable for testing.
2. Long test sequences. A test for a fault in the sequential logic essentially contains 3 parts:
   (a) Initialisation of the internal memory.
   (b) A combinational test to activate the fault and to bring its effect to the boundary of the combinational logic.
   (c) If the fault has affected one or more memory elements, then the state observation of one of the affected elements at a primary output.

The test for a fault in a sequential circuit may be a sequence of several vectors that must be applied in the specified order. One simple TPG technique applied for synchronous circuits is an iterative (time-frame expansion) TPG method using a TPG algorithm for combinational logic. The basic idea is to divide the circuit structure to several time frames. It means we receive several copies of the same combinational circuit with not only primary inputs and outputs but also with pseudo-primary inputs (outputs from flip-flops) and pseudo-primary outputs (inputs to flip-flops). The fault has to be injected in the same node in all time frames.

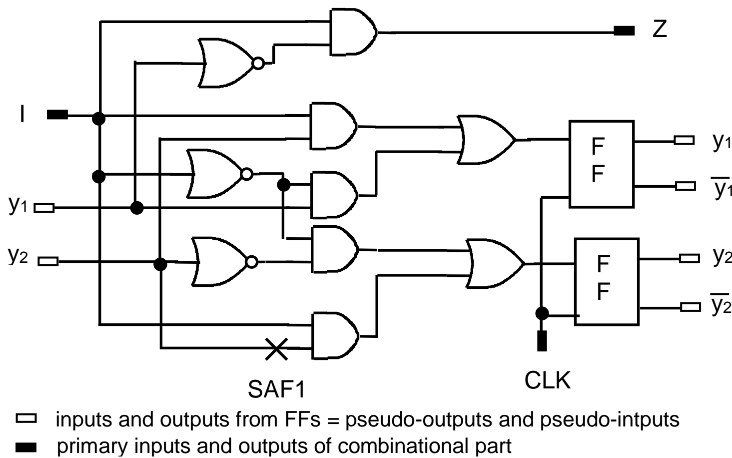


Figure 3-16. Sequential circuit for *Example 3-2*

The major problem is that the number of circuit structure copies is not known at starting the test generation process. The number of copies depends on the FFs states for the fault propagation to a primary output. Test generation for complex sequential circuits is a time consuming and memory space demanding process. No clock signal faults and internal faults of flip-flops are modelled in this TPG method.

Example 3-2: Consider test sequence generation for the sequential circuit presented in *Figure 3-16* and injected SAF1 with initial FF states (y_1y_2) = (11) using the iterative TPG technique. The circuit structure is divided into time frames according to the structure presented in *Figure 3-17*. The test sequence I = (1,1,1) for SAF1 is generated in 3 time frames where q_1^+ and q_2^+ mean the next state of FFs.

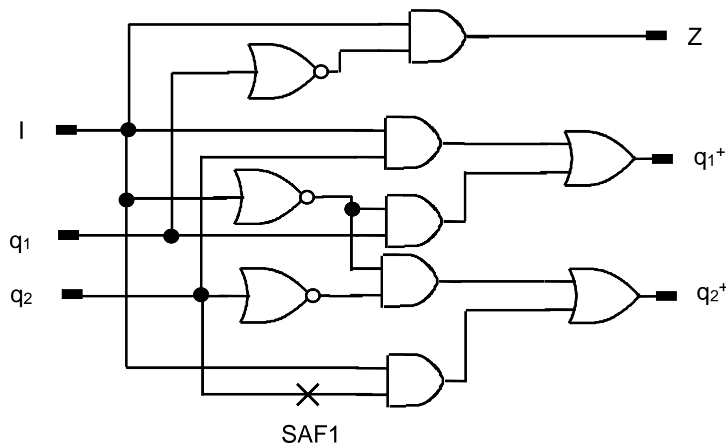


Figure 3-17. One time frame for sequential circuit in *Figure 3-16*

Other TPG methods are simulation-based methods [1], [2], [15]. Nowadays complex sequential circuits are designed by scan design methods (described in Chapter 4) to avoid the TPG problems for sequential logic and ATPG is applied only for combinational parts, and sequential parts are tested as a scan chain by the flush test (1100110011....).

Test set for delay faults requires a 2-pattern ordered test set. The first pattern is devoted to initialisation of a specific value in a DUT node and the second one for delay faults excitation. The test set is categorised as robust or non-robust test set. A robust test detects the targeted delay faults irrespective of the presence of other delay faults in the DUT. Otherwise, the test set is non-robust. TPG algorithms for path delays use a model with multiple logic values (e.g. 5,9,11,13 values). TPG methods for transition faults are mostly

based on the principle that test vectors for SAFs can be used for constructing the test set for delay faults [2],[14].

The efficient TPG algorithms are the central part of each ATPG process. The strength of the ATPG rests in its algorithms for tracing through the design description and establishing values. Therefore TPG algorithms are still one of the hot research tasks. The overall test set preparation includes other tasks that have to be done before and after the application of the ATPG tool. They include preparing the computer environment for the tool, preparing the tool for accepting the design description using during structure analysis and test vectors processing and to assign with the data and control format of the tester.

3.1.2 Test generation with BDDs

As the complexity of digital systems continues to increase, the gate level test generation methods become obsolete. Promising approaches are high-level, multi-level or hierarchical methods which use behavioral, functional or multi-level descriptions of systems. Nevertheless a uniform approach to test generation at different levels is missing, a lot of different languages, models and formalisms depending on the level are used.

Decision Diagrams (DD) can serve as a basis for a uniform approach to test generation for mixed-level representations of systems, similarly as we use the Boolean algebra for the plain logic level. In the following we describe how the traditional logic level test generation methods can be implemented on **Binary Decision Diagrams** (BDD) [23], [24], [25], [29] as a special class of DDs, and then we generalize the procedures developed for BDDs for a general class of DDs [27], [28], [31] to handle the test generation problems at higher levels of systems.

Structurally synthesized BDDs. In 1959 C.Y.Lee introduced a method for representing digital circuits by Binary Decision Programs [23]. In 1976 the same model called alternative graphs [24] was introduced for test generation purposes. Independently the same model was introduced into the field of test generation by Akers [25] under the name of Binary Decision Diagrams (BDD). Today the theory of BDDs is developing quickly [29], [30], [32].

In [24], [26], [31], [33], [34] **structurally synthesized BDDs** (SSBDD) as a special class of BDDs was introduced to represent the topology of gate-level circuits in terms of signal paths. Unlike “traditional” BDDs, SSBDDs [29], [30], [32] directly support test generation for gate-level structural faults without representing these faults explicitly. The advantage of the SSBDD based approach is that the library of components is not needed for structural path activization. This is the reason why SSBDD based test generation procedures do not depend on whether the circuit is represented on the gate

level or on the macro-level whereas the macro means an arbitrary single-output subcircuit of the whole circuit. Moreover, the test generation procedures developed for SSBDDs can be easily generalized for **higher level DDs** to handle digital systems represented at higher levels [27], [28], [31].

The BDD that represents a Boolean function is a directed noncyclic graph with a single **root node**, where all **nonterminal nodes** are labelled by Boolean variables (arguments of the function) and have always exactly two successor-nodes whereas all **terminal nodes** are labelled by constants 0 or 1. For all nonterminal nodes, a one-to-one correspondence exists between the values of the label variable of the node and the successors of the node. This correspondence is determined by the Boolean function inherent to the graph.

Denote the variable which labels a node m in a BDD by $x(m)$. We say that a value of the node variable activates the node output edge. According to the value of $x(m)$, one of two output edges of m will be activated. If $x(m)=1$ we say 1-edge is activated, or if $x(m)=0$ we say 0-edge is activated. A **path in a BDD** is activated if all the edges that form this path are activated. The **BDD is activated** to the value 0 (or 1) if there exists an **activated path** which includes both the root node and the terminal node labelled by the constant 0 (or 1).

Definition 3.1. A BDD G_y with nodes labelled by variables $x_1, x_2, ..., x_n$, represents a Boolean function $y = f(X) = f(x_1, x_2, ..., x_n)$, if for each pattern of X, the BDD will be activated to the value which is equal to y.

Important property of SSBDDs. SSBDDs differently from traditional BDDs have the following property: each node m in a G_y which describes a tree-like subnetwork N_y of the gate-level circuit N, represents a signal path $l(m)$ in N_y.

An example of a combinational circuit with a tree-like **macro** and SSBDD for the macro is presented in *Figure 3-18*. For simplicity, the values of variables on edges of the SSBDD are omitted (by convention, the 1-edge is always directed to the right, and the 0-edge is always directed downwards). Also, terminal nodes with constants 0 and 1 are omitted (leaving the SSBDD to the right corresponds always to $y = 1$, and down - to $y = 0$). Each node is marked by an input variable of the macro. A node with the label x_m in the SSBDD represents the signal path through the macro which begins with the input variable x_m. The node variable is inverted when the path consists of odd number of inverters, and not inverted when the number of inverters is even. For example, the node $x_{7,1}$ of SSBDD represents the **signal path** with even number of inverters starting with the line $x_{7,1}$, through the nodes a,d,e to the output y in the macro (the bold lines in the circuit). The node x_1 in the SSBDD is inverted since the corresponding path x_1,d,e,y consists of odd number of inverters. The fan-out node x_7 in the

circuit has three branches, and each branch $x_{7,i}$, $j = 1,2,3$ is the beginning of a path which is represented by the node $x_{7,i}$ in the SSBDD.

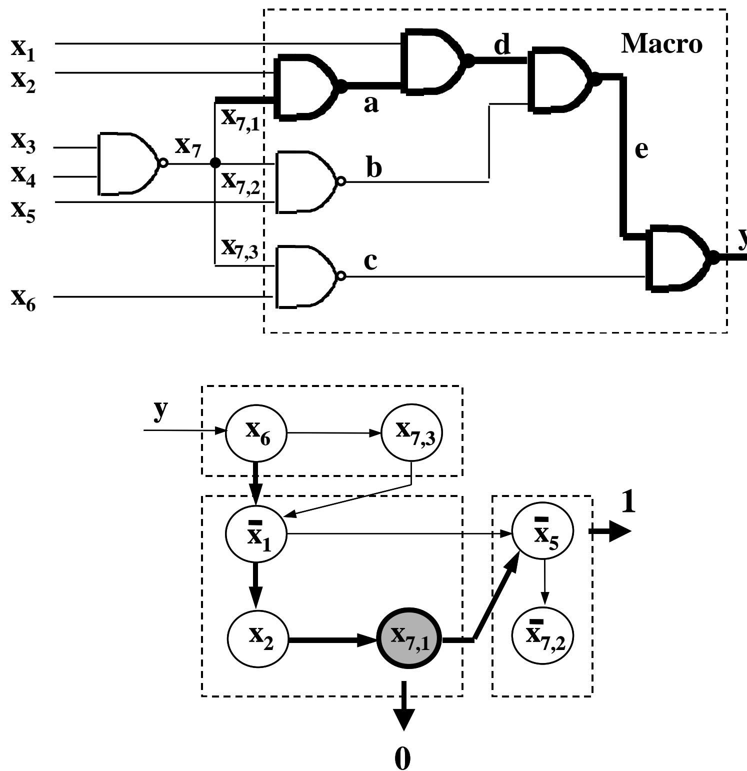


Figure 3-18. Combinational macro and his SSBDD

The one-to-one correspondence between nodes *m* in a SSBDD and paths *l(m)* in the corresponding gate-level circuit is the direct result of the synthesis procedure of SSBDDs.

From the above-described property of the SSBDD, automatic **fault collapsing** results. Assume a node *m* with label variable *x(m)* represents a signal path *l(m)* in a circuit. Suppose the path *l(m)* goes through *n* gates. Then, instead of 2*n* faults of the path *l(m)* in the circuit, only 2 faults related to the node variable *x(m)* should be tested when using the SSBDD model.

Generation of SSBDDs. For synthesis of SSBDDs for a **gate network**, the graph superposition procedure is used [24], [31]. If the label *x*(*m*) of a node *m* in the SSBDD G_y is an output of a subnetwork which is represented by another SSBDD $G_{x(m)}$ then the node *m* in G_y can be substituted by the graph $G_{x(m)}$. In this **graph superposition** procedure the following changes in G_y and $G_{x(m)}$ are made.

<u>Algorithm 3-1</u>. Graph superposition

1) The node m will be removed from G_y.
2) All the edges in $G_{x(m)}$ that were connected to terminal nodes $m^{T,e}$ in $G_{x(m)}$ will be cut and then connected, correspondingly, to the successors m^e of the node m in G_y. Here, $m^{T,e}$ is the terminal node labelled by constant $e \in \{0,1\}$
3) All the incoming edges of m in G_y will be now incoming edges for the initial node m_0 in $G_{x(m)}$.

Consider a gate-level description of a network where each gate is represented by a BDD. Starting from the BDD of the output gate, and using iteratively the superposition procedure, we can compress the initial model of the gate-network (by each substitution we reduce the model by one node and by one graph). To reach high **compression** (to reduce **complexity**) of the model, we generate SSBDDs only for **tree-like subnetworks**. As the result we get a **macro network** where each macro (a tree-like subcircuit) is represented by a SSBDD.

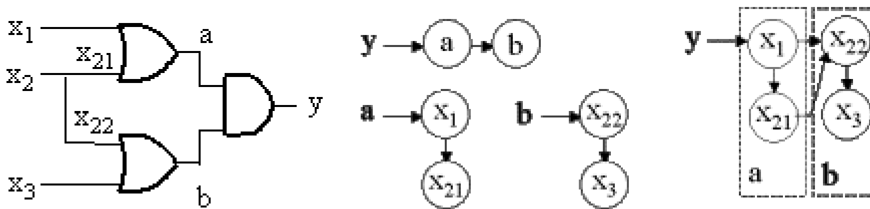


Figure 3-19. Example of superposition of graphs

<u>Example 3-3</u>

An example of the graph superposition procedure is shown in *Figure 3-19*. We start with the output AND gate, and its BDD G_y which consists of two nodes a and b. The input a of the AND gate is simultaneously the output of the OR gate represented by the BDD G_a which consists of the nodes x_1 and x_{21}. First, we substitute the node a in G_y by the graph G_a. Thereafter the node b in G_y is substituted by the graph G_b which consists of the nodes x_{22} and x_3. The final graph which represents the whole circuit consists of the nodes x_1, x_{21}, x_{22}, and x_3.

Test generation with SSBDDs. Consider a **combinational circuit** as a network of gates, which is partitioned into interconnected tree-like subcircuits (macros). This is a new higher level (macro-level) representation of the same circuit. Each macro is represented by a SSBDD where each node corresponds to an input of the macro. In the tree-like subcircuits only the

stuck-at faults at inputs should be tested (see Section 1.2 in Chapter 2). This corresponds to testing all the nodes in each SSBDD.

Test generation for a node m in SSBDD, which represents a function $y = f(X)$ of a tree-like subcircuit (macro), is carried out by the following procedure [31], [33], [34].

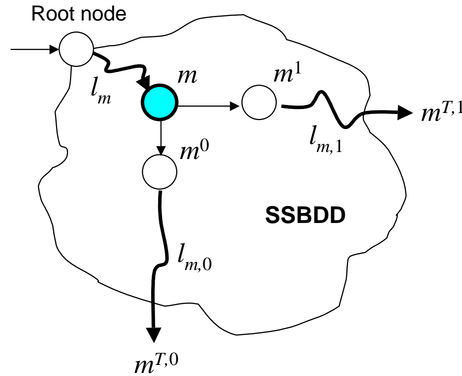


Figure 3-20. Test generation for the node *m* with SSBDD

<u>*Algorithm 3-2.*</u> Test generation for a node *m* in the SSBDD G_y, $y = f(X)$

1) A path l_m from the root node of SSBDD to the node *m* is activated.
2) Two paths $l_{m,e}$ consistent with l_m, where $e \in \{0,1\}$, from the neighbors m^e of *m* to the corresponding terminal nodes $m^{T,e}$ should be activated. If the node *m* is directly connected via *e*-edge to $m^{T,e}$ no path $l_{m,e}$ should be activated for this particular value of *e*.
3) For generating a test for a particular stuck-at-*e* fault $x(m) \equiv e$, $e \in \{0,1\}$, the opposite assignment is needed: $x(m) = \bar{e}$.
4) All the values assigned to node variables (to variables of *X*) build the local test pattern $T(X,y)$ (input pattern of the macro) for testing the node *m* in G_y (for testing the corresponding path $l(m)$ on the output *y* of the given tree-like circuit).

The paths in the SSBDD activated by the described procedure are illustrated in *Figure 3-20*.

To create the final test pattern in terms of primary inputs of the circuit for the given fault in an embedded macro of the circuit, similar fault propagation and line justification procedures are needed as described in Subsection 3.1. The difference, however, is that these procedures will be carried out on the higher macro (instead of the gate) level whereas the macros of the circuit are represented by SSBDDs. The fault propagation through a macro from the input *x* to its output *y* is carried out similarly to the test generation for the node *m* labelled by *x* in the corresponding SSBDD G_y as explained in *Algorithm* 3-2. Line justification for the task $y = e$ is carried out by activating a path in the graph G_y from the root node to the terminal node $m^{T,e}$.

Example 3-4

Consider test generation for the fault SAF0 on the internal branch (input of the macro) $x_{7,1}$ in the circuit in *Figure 3-18*. For SSBDD we have to generate by *Algorithm 3-2* a test for the node $x_{7,1}$ at the condition $x_{7,1} = 1$. Activating the path l_m through the nodes x_6, x_1, and x_2 gives assignment x_6=0, x_1=1, and x_2=1. Activating the path $l_{m,1}$ through the node x_5 gives x_5=0. The path $l_{m,0}$ is activated "automatically", since the 0-edge from the node $x_{7,1}$ is connected directly to the terminal node $m^{T,0}$. The paths, activated by test pattern $x_1x_2x_3x_4x_5$ = 11001, are shown by bold lines in *Figure 3-18*.

In general we represent a circuit by a network of tree-like subcircuits where each subcircuit is represented by a SSBDD. To generate a full test for a given circuit, test patterns should be generated for SAF faults of all nodes in all SSBDDs. The procedure of test generation for a chosen fault consists of three steps - **fault senzitization**, **fault propagation** and **line justification** as explained in Section 3.1.1 for the gate-level approach.

Fault sensitization. Sensitizing a fault on a line means to assign the complement of the faulty signal on the line. Since the faulty signal may be 0 or 1, we distinguish it by denoting the SAF0 (SAF1) fault with D ($\overline{D}$) analogically to the classical D-algorithm where D $\in \{0,1\}$[1], [2].

Consider a graph G_y representing a tree-like subcircuit with Boolean function $y = f(X)$ where X is the vector of input variables of the subcircuit. Choose a node m labelled by an input variable $x(m) \in X$ in G_y for generating a test pattern for the fault SAFe, $e \in \{0,1\}$. To sensitize the fault we generate a local test pattern for testing m with condition $x(m) = \overline{e}$ as explained above. Assign to y a symbolic value D if y = 1 in the generated local test pattern $T(X,y)$ or $\overline{D}$ if y = 0. This symbolic value should be propagated to one of the outputs of the circuit. All the generated input values of X should be justified by primary input values of the circuit.

Fault propagation. In general, the fault propagation procedure on SSBDDs is similar to the test generation on a single SSBDD. To propagate a symbolic value D ($\overline{D}$) from x to y in a SSBDD G_y where $y = f(X)$ and $x \in X$, we have to find in G_y a node m labelled by x and generate a local test pattern for m at the given constraints. The constraints are all the previously assigned values of the variables of the circuit.

The new technique compared to fault sensitization is related to handling the symbolic value D when tracing the paths on SSBDDs. Two techniques may be used: **single path activation**, or **multiple path activation** (similar to the well known D-algorithm [1], [2]).

In single path activation we propagate the value D along a single path in a circuit and block its propagation along all other parallel paths. On SSBDDs it means that during path activation procedures we have to treat all assigned values D as unknown values that should remain unknown. In other words

when tracing and activating the paths l_m (or $l_{m,e}$) on a SSBDD through a node m' with symbolic value $x(m') = D$ we have to reach the target node m (or $m^{T,e}$) from both outputs of the node m'. As the result, the activation of the path l_m or ($l_{m,e}$) will not depend on the changing value of $x(m')$.

Example 3-5

Consider a circuit and its SSBDD in *Figure 3-21*. Let us propagate the value D from the input x_2 through the branch $x_{2,2}$ in a circuit, and block its propagation through the branches $x_{2,1}$ and $x_{2,3}$. On the SSBDD the task is equivalent to generation of a test for the node $x_{2,2}$. When activating the path $l_{m,1}$ we meet a node with unknown value of $x_{2,3}$. So, we have to reach the terminal node 1 for both cases $x_{2,3} = 0$, and $x_{2,3} = 1$. For that we have to assign $x_3 = 1$. On the other hand, when activating the path $l_{m,0}$ we meet a node with unknown value of $x_{2,1}$. So, we have to reach the terminal node 0 for both cases $x_{2,1} = 0$, and $x_{2,1} = 1$. For that we have to assign $x_1 = 0$. Now the value D is propagated from x_2 through the circuit to y via a single path (via edge $x_{2,2}$). The resulting test pattern is $x_1, x_2, x_3 = 0D1$.

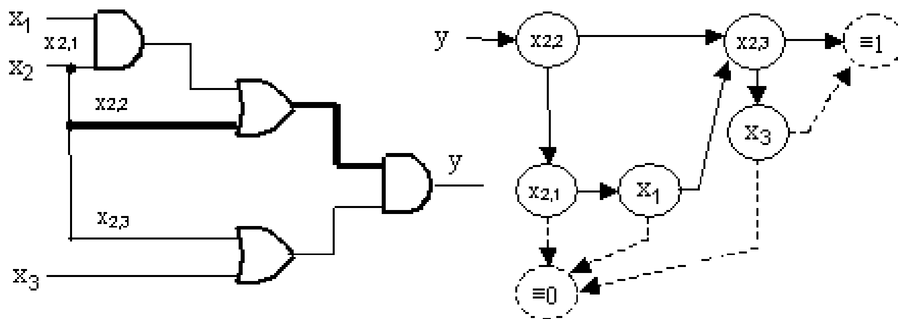


Figure 3-21. Multiple path activation by using SSBDDs

To allow *multiple path activation* on SSBDDs (more general case) we can exploit properly the duality of the symbolic value $D \in \{0,1\}$ in the path activation. Suppose the fault should be propagated through the path represented on SSBDD by a node m. Let us have $x(m) = D$ (or $\overline{D}$). To propagate D via multiple paths through the circuit we will use *Algorithm 3-2* in a slightly modified form: instead of activating the three separate paths l_m, $l_{m,1}$, and $l_{m,0}$, we will activate now two overlapping composite paths $l_m.l_{m,1}$ and $l_m.l_{m,0}$. When tracing and activating the path $l_m.l_{m,1}$, the value D is taken as 1 (if $x(m) = D$) or 0 (if $x(m) = \overline{D}$), and in the opposite case, when tracing and activating the path $l_m.l_{m,0}$, the value D should be taken as 0 (if $x(m) = D$) or 1 (if $x(m) = \overline{D}$).

Example 3-6

Consider again the circuit and its SSBDD in *Figure 3-21*. Propagate the value D through all the three paths in a circuit. In the SSBDD the task is equivalent to generating a test for the node $x_{2,2}$ with taking D = 1 on the path $l_{m,1}$ and taking D = 0 on the path $l_{m,0}$. It is easy to see that no additional assignments are needed for propagating the fault from x_2 to y when using the SSBDD model. The resulting test pattern is: x_1, x_2 , x_3 = *-D-* , where dash means *don't care*. When trying to activate multiple paths in the gate level we would need two additional assignments: $x_1 = 1$ and $x_3 = 0$ with resulting test pattern: x_1, x_2 , x_3 = 1*D*0. By keeping the variables x_1, x_3 free (not assigned) we can avoid many backtracks later in searching the test, and the test generation speed will increase.

The last example shows another advantage of using SSBDDs in test generation compared to the gate-level approach. The explanation of this effect results from the fact that we work only with input variables of tree-like subcircuits instead of the internal variables of gate-level representation.

3.2 HIGH-LEVEL TEST GENERATION

Since traditional gate-level test generation algorithms for complex VLSI systems have lost their importance, other approaches based mainly on functional, behavioral, or hierarchical methods are gaining popularity. Functional test generation methods [1], [2], [35], [36], [37], [38], which do not use low-level implementation data, help increase the speed of test generation, however, they cannot achieve good test quality measured in terms of gate-level fault coverage. Hierarchical methods [39], [40], [41], [42] take advantage of the high level information for speeding up test generation while providing good coverage of low level faults or physical defects.

3.2.1 Overview of methods for high level test generation

Functional test generation without fault model. Structural details are not often given for complex digital systems. In these cases, **functional test generation** is used. Since the **functional model** of a system is independent on the implementation, the functional tests derived from functional models can be used not only to check whether physical faults are present in the circuit or system, but also as a design verification tool with which we check whether the implementation is free of design errors.

Functional test generation can be done in two different ways [2]: by using specific functional fault models (Chapter 2), or by trying to derive tests with the knowledge of the specified fault-free behavior only.

In the first case heuristic and formal approaches can be used. BDD models as the specification of a system have been suggested in formal approaches [25], [43]. The BDDs can be still used for not very complex circuits only, and hence, can have only restricted use because of an exploding complexity of the model.

Heuristic or ad hoc functional testing methods try to exercise all the functions of the system. In case of the microprocessor, a typical functional test is based on exercising the instruction set in a specific order. An important issue is whether the instruction set is orthogonal [2]. An **orthogonal instruction set** allows every operation to be executed in every possible addressing mode. This feature means that the operation decoding and address computation are independent, and these mechanisms can be tested separately. If the instruction set is not orthogonal, then every operation must be tested with all its addressing modes which leads to a very long test sequence.

Functional testing can be optimized by using the start-small (or bootstrap) approach [2], in which the tests performed at a certain step use components and/or instructions tested in the previous steps. In this way the tested part of the system is gradually extended.

A technique for ordering the instructions of a microprocessor according to the **start-small principle** is presented in [44]. The **cardinality** of an instruction is defined as the number of registers accessed during the execute phase of the instruction. Instructions are tested in increasing order of their cardinality. In this way the instructions affecting fewer registers are tested first. Among instructions of the same cardinality, priority is given to those with higher **observability**.

As mentioned already, the disadvantage of functional approaches of test generation is the low fault coverage measured at the implementation level (e.g. low gate level stuck-at fault coverage).

Test generation with symbolic execution trees. In Chapter 2 an RT level fault model was described. It can be used for formalized fault model based test generation. It can be justified that the functional faults of that model manifest physical faults of lower levels (SAF, bridging faults etc.) as functional faults at the RT level.

Test generation for a complex digital system requires a concise description of the functions of the system. We have to describe all the possible disjoint **modes of operation** of the system by giving the input conditions and observing the effects of each mode. **Hardware description languages** can be classified into **procedural** and **nonprocedural** languages. The key difference is in the way of handling sequencing of activities. Procedural descriptions are easier to write, understand, and verify than nonprocedural descriptions. However, nonprocedural descriptions express

the semantics of hardware function more directly than the procedural descriptions, and they are therefore more suitable for fault model based test generation. **Symbolic execution** of procedural descriptions can be used to bridge the gap between the two types of descriptions [45]. This technique has been used to prove the correctness of machine architectures implemented in microcode.

The description of a system is usually given as a **set of RT level statements**. An example of a RT level statement and its **symbolic execution tree** as the result of symbolic execution of the statement is presented in *Figure 3-22.* Each path in the tree represents a particular working mode of the system. The list of all paths of the tree can be used by any test generation procedure as a checklist to generate the test patterns required to exercise each mode of operation.

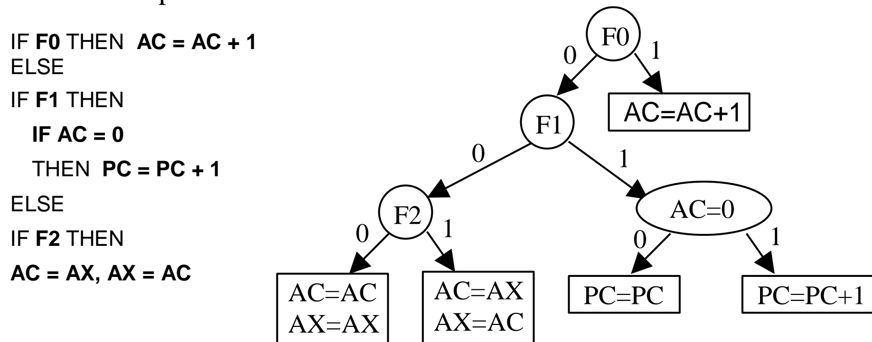


Figure 3-22. Symbolic execution tree for a function submodule

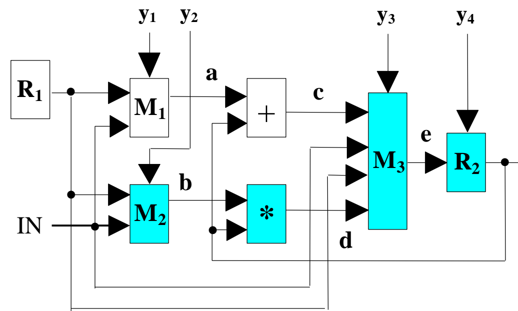
The test generation technique based on symbolic execution trees involves the following steps [46].

Algorithm 3-3. Test generation with symbolic execution trees

1) Derive test generation order for the set of function sub-modules in the whole RTL description (an instruction of a microprocessor may represent such a function submodule);
2) Using the order obtained for every function sub-module, set up its symbolic execution tree for terminated paths;
3) Perform heuristic test procedures derived from the results of Step 2 for data storage and data transfer faults in the current function sub-module;
4) Inject an RTL fault which has not been tested along the selected path;
5) Set up the symbolic execution sub-tree of the fault-injected machine and choose one terminated path for faulty symbolic results;
6) Derive a test pattern for current fault by comparing the symbolic results and path constraints of good and bad machines.

In testing complex digital systems (e.g. microprocessors), an order of testing based on the partition of function modules (e.g. instructions) is needed to efficiently perform the formalized test generation [44]. Each RTL statement in each instruction is tested comprehensively based on RTL fault model.

Structural register transfer level test generation. At higher levels a design is represented as a network of higher-level components. For example, at the RT level, a circuit can be viewed as an interconnection of components



like registers, counters, multiplexers, adders, etc. Knowing how these components function allows using simple solutions for line-justification, fault-propagation or implication tasks similarly like in the D-algorithm [2].

Figure 3-23. RT level data-path

As an example, in *Figure 3-23* an RTL data-path is presented. Potential RT level functions of the components are represented in *Table 3-3* (for simplification, the control signals are represented by integer variables).

Table 3-3. Potential operations of the components in *Figure 3-23*

M_1		M_2		M_3		R_2		
y_1	Function	y_2	Function	y_3	Function	y_4	Operation	F
0	$M_1 = R_1$	0	$M_2 = R_1$	0	$M_3 = M_1 + R_2$	0	Reset	$R_2 = 0$
1	$M_1 = IN$	1	$M_2 = IN$	1	$M_3 = IN$	1	Hold	$R_2 = R'_2$
				2	$M_3 = R_1$	2	Load	$R_2 = M_3$
				3	$M_3 = M_2 * R_2$			

The word variables R_1, R_2 and R_3 represent registers, the integer variables y_1, y_2 , y_3, and y_4 represent the control signals. M_1, M_2 and M_3 are multiplexers. There is also an adder and a multiplier in the circuit.

For *line justification*, we first determine the set of potential operations the component can execute, and among these we then select one that can produce the desired result. Assume we wish to justify R_2 = 1110. From the values of the control variable we determine that the potential operations are

{Reset, Hold, Load}. Reset and Hold are not possible. For the Load operation we have the solution: $y_4 = 2$, $M_3 = 1110$.

For *fault propagation*, we have to define corresponding working modes of the components in order to allow propagation of erroneous signals from inputs to outputs. Denote by D a symbolic faulty value of a vector variable to be propagated to an observable node of the RT level circuit. As an example, the fault propagation modes which make the component M_3 transparent and allow propagation of erroneous signals from the inputs to the output are given in *Table 3-4*.

Table 3-4. Transparent modes of M_3 in *Figure 3-23* for fault propagation

y_3	M_1	M_2	R_1	R_2	IN	M_3
0	*D*	x	x	0	X	*D*
3	x	*D*	x	1	X	*D*
2	x	x	*D*	x	X	*D*
0	0	x	x	*D*	X	*D*
3	x	1	x	*D*	X	*D*
1	x	x	x	x	*D*	*D*

Hierarchical approach to test generation for systems. In **hierarchical test generation** for digital systems, top-down and bottom-up strategies are well-known [39-42], [47], [48]. In the **bottom-up approach**, local test patterns of components pre-calculated at the lower level are assembled into the test frames generated at the higher abstraction level. Generality of the approach is in the possibility of using **precalculated library tests** for components during higher level test planning. On the other hand, such algorithms typically ignore the incompleteness of the problem: **high-level constraints** imposed by other modules and/or the network structure may prevent test vectors from being assembled. This can cause that solutions in test generation cannot be found even if they exist.

The **top-down approach** has been proposed to solve this problem by deriving environmental constraints for low-level solutions. However, the complex nature of high-level constraints makes it difficult to consider them in low level test generation. Also, such technique may be of little use when the system is still under development in a bottom-up way, or when the pre-generated local tests have to be applied.

In the high-level test generation (either for bottom-up or top-down approaches), the test properties of system components (modules) are described in the form of **fault-propagation modes** (see *Table 3-4*). These modes usually are defined either by lists of control signals such that the data on input lines are reproduced without logic transformation into the output lines (called **I-path** [47]), or by a list of control signals that provide one-to-

one mapping between data inputs and data outputs (**F-path** [48]). The I-paths and F-paths constitute connections that can be used to propagate test vectors from input ports to the module inputs and to propagate the test response to an output port.

In this approach, fault-propagation modes and the library test sets typically constitute two separate description packages for modules. The approach may be useless if not all functions in multi-functional modules are used in the given application, because in that case not all library test patterns can be assembled at the higher level, which results in a reduced fault coverage. Also, using local test patterns and only fault-propagation modes is not sufficient for high-level (behavioral) test generation, because the functional description of modules needed for test generation is only partially represented in this information.

In the next subsection, a method is presented, which allows implementing both bottom-up and top-down approaches at uniform basis. We start with the bottom-up approach. The transparency features will be exploited to build up transparent I- or F-paths for assembling library patterns of components. In these cases when the bottom-up approach fails (no transparent path can be activated), we switch to the top-down approach for extracting high-level constraints with the goal to be considered when deriving tests for components at the lower level.

3.2.2 Test generation for digital systems with decision diagrams

Representing digital systems with high-level decision diagrams. Test generation methods developed for SSBDDs have an advantage compared to other logic level methods, namely that they can be easily generalized to handle the test generation problems at higher levels of systems [27], [28], [31].

In general case beyond the Boolean algebra a decision diagram can be defined as a non-cyclic directed graph $G = (M,\Gamma,X)$ with a set of nodes M, a set of variables X, and a relation Γ in M where $\Gamma(m) \subseteq M$ denotes the set of successors of the node $m \in M$ [31]. The nodes $m \in M$ are labelled by variables $x(m) \in X$ (constants or algebraic expressions of $x \in X$). For each value e from a set of possible predefined values $e \in V(x(m))$ of a non-terminal node variable $x(m)$, there exists a corresponding output edge from the node m into a successor node $m^e \in \Gamma(m)$. Consider a situation where all variables $x \in X$ are fixed to particular values. By these values, for each non-terminal node m a certain output edge is chosen, which is connected to a successor node. Let us call these connections between nodes - **activated edges**, and the chains of them - **activated paths**. For each pattern of values

of $x \in X$, there exists always a **full activated path** from the root node to a terminal node. This relation describes a mapping from a Cartesian product of the sets of values $V(X)$ for variables $x \in X$ in all nodes to the joint set of values $V(Y)$ of expressions in terminal nodes. Therefore, by DDs it is possible to represent arbitrary digital functions $Y=F(X)$, where Y is the variable whose value will be calculated by the DD and X is the vector of all variables in the nodes of the DD.

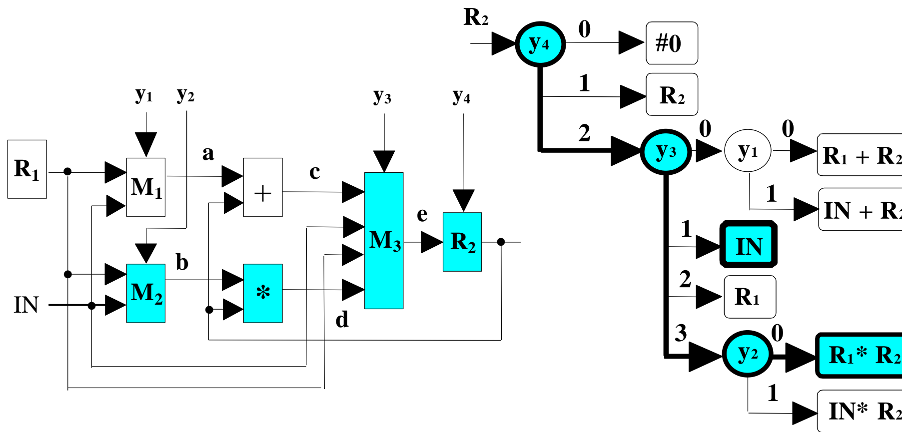


Figure 3-24. Register-transfer level data-path system

Depending on the type of the system (or its representation level), we may have various DDs, where nodes have different interpretations and relationships to the system structure. In register transfer level (RTL) descriptions, we usually partition a digital system into the **control part** and the **data part**. State and output variables of the control part serve as addresses and control words, and the variables in the data part serve as data words. High-level data word variables describe RTL functions in data parts. When using DDs to describe complex digital systems, we have to represent the system by a suitable set of interconnected components (combinational or sequential subcircuits). Thereafter, we have to describe these components by their corresponding functions which can be represented by DDs.

Figure 3-24 shows a RTL data-path and its DD [49]. The DD is created by superposition of elementary DDs of components of the circuit, analogically to the superposition procedure for SSBDDs described in Subsection 1.2. The word variables R_1, R_2 and R_3 represent registers, the integer variables y_1, y_2, y_3, and y_4 represent the control signals. M_1, M_2 and M_3 are multiplexers, and the functions R_1+R_2 and R_1*R_2 represent the adder and multiplier, respectively. The whole DD describes the behavior of the input logic of the register R_2.

In test pattern simulation, a path is traced in the graph, guided by the values of input variables until a terminal node is reached, similarly as in the case of SSBDDs. In this example, the result of simulating the vector y_1, y_2, y_3, y_4, R_1, R_2, IN = 0,0,3,2,10,6,12 is $R_2 = R_1*R_2 = 60$ (bold arrows mark the path activated by the control pattern). Instead of simulating all the components in the circuit by a traditional approach, on the DD only 3 control variables are visited during simulation, and only a single data manipulation $R_2 = R_1*R_2$ is carried out.

Each node in a DD represents a subcircuit of the system. For example, the nodes y_1, y_2, y_3, y_4 represent multiplexers and decoders, the nodes R_1, R_2, IN, and other terminal nodes represent registers, input bus, and data manipulation subcircuits, respectively. To test a node of the DD means to test the corresponding subcircuit.

Test generation for RT level data paths with a single DD. RT level data path can be represented by a single DD as shown in *Figure 3-24*. A test for such a system can be created in two parts [31]:

- **conformity test**, which makes sure that the different working modes chosen by control signals are properly carried out, and
- **scanning test**, which makes sure that the different functional blocks are working correctly.

The task of the conformity test is to detect the control faults and the faults in multiplexers. In terms of DDs the non-terminal nodes are tested by the conformity test. For creating the conformity test we may use either high-level fault models or the hierarchical approach.

The task of the scanning test is to detect the faults in registers, buses and data manipulation blocks. In terms of DDs the terminal nodes are tested by the scanning test. For creating scanning test sequences hierarchical test generation approach is preferred, since no good high-level fault models for testing data manipulation blocks are known. In simpler cases (like buses and registers) pure high-level or functional test generation approaches may still be used.

Conformity test. Consider a nonterminal node m labelled by a control variable $x(m)$ in the given DD G_Y representing a digital system with a function $Y = F(X)$. Let $X = (X_C, X_D)$ where X_C is the vector of control variables and X_D is the vector of data variables. To generate a test for the node m means to generate a test for the control variable $x(m) \in X_C$. Suppose that the variable $x(m)$ may have $n = |V(x(m))|$ different values. For testing $x(m)$, we have to activate and exercise all the proper working modes controlled at least once by each value of $x(m)$. For each of such a working mode, the needed state of the system should be generated, so that every possible faulty change of $x(m)$ should produce a faulty next state, which differs from the expected next state of the given working mode.

Denote by $m^e \in \Gamma(m)$ the successor node of the node m for the value $x(m) = e$, where $e = 1,2,\ldots,n$. For generating a test for m we have to solve the following tasks on the DD (*Figure 3-25*).

<u>*Algorithm 3-4.*</u> Conformity test generation for a nonterminal node m

1) Activate a path l_m from the root node of the DD up to the node m by choosing proper values for all the variables in the nodes of l_m.
2) For each $e = 1,2,\ldots,n$ activate consistent non-overlapping paths $l_{m,e}$ from m^e up to a terminal node $m^{T,e}$ for all successor nodes m^e of m by choosing proper values for all the node variables on all the paths $l_{m,e}$.
3) Find the proper set of data (the values of the variables in X_D), so that the inequality

$$f(m^{T,1}) \neq f(m^{T,2}) \neq \ldots \neq f(m^{T,n})$$

holds, where $f(m^{T,e})$ is the functional expression in the terminal node $m^{T,e}$ reached by the path $l_{m,e}$.

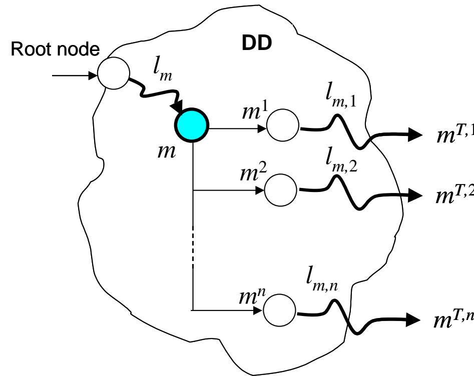


Figure 3-25. Conformity test generation on the DD

Consider the resulting symbolic test pattern in the following symbolic way:

$$T = eCD;Y_e$$

where e is the symbolic value of the tested variable $x(m)$, C is the vector of other control signals $X'_C \subseteq X_C$ generated in the first two steps of the algorithm, D is the vector of data values generated in the third step of the algorithm, and Y_e is the expected output value of the circuit corresponding to the value e of the tested control variable $x(m)$. The final conformity test of the control variable $x(m)$ created from the generated symbolic test pattern $T = eCD;Y_e$ can be implemented as the following test program:

<u>*Algorithm 3-5.*</u> Conformity test implementation for $T = eCD;Y_e$

For each value of $e = 1,2, \ldots, |V(x(m))|$
BEGIN

Load the data registers with D
Carry out the working mode defined by the value of e
- at the generated control signals $x(m) = e$, $X'_C = C$, and
- at the generated input data signals

Read the value of Y_e.
END.

Example 3-7

Generate a test program for testing the multiplexer M_3 represented by the node m with label y_3 in *Figure 3-24*. We activate 4 paths $l_{m,e}$ for each value e = 0,1,2,3 of y_3. Two of them, $l_{m,1}$, $l_{m,2}$, for values $y_3 = 1$ and $y_3 = 2$, respectively, are "automatically" activated since the successors of the node y_3 for these values are terminal nodes. The control values for the test are found by activating the path l_m with assigning $y_4 = 2$, and by activating two paths $l_{m,0}$ and $l_{m,3}$ with assigning $y_1 = 0$ and $y_2 = 0$, respectively. The test data D: $R_1 = D_1$, $R_2 = D_2$, $IN = D_3$ are found by solving the inequality

$$R_1 + R_2 \neq IN \neq R_1 \neq R_1 * R_2.$$

The following conformity test program for the control variable y_3 results:

For e = 1,2,3,4
BEGIN
Load the data registers $R_1 = D_1$, $R_2 = D_2$
Carry out the tested working mode at $y_3 = e$, $y_1 = 0$, $y_2 = 0$, $y_4 = 2$ and $IN = D_3$
Read the value of $R_{2,e}$.
END.

Scanning test. Consider a terminal node m^T labelled by a functional expression $f(m^T)$ in the given DD. To generate a test for the node m^T means to generate a test for the function $f(m^T)$. Denote by $X(m^T) \subseteq X_D$ the data variables used in the expression $f(m^T)$.

Test generation process for testing $f(m^T)$ is carried out according to the following algorithm.

Algorithm 3-6. Scanning test generation for a terminal node m^T

1) Activate a path $l_{m,T}$ from the root node of the DD up to m^T by choosing proper values C for all the control variables in the nodes of $l_{m,T}$.
2) Find the proper sets of data values $D = (D_1, D_2, \ldots, D_n)$ of $X(m^T)$ for testing the function $f(m^T)$ (this operation can be carried out at the lower (e.g. gate) level if the implementation details for $f(m^T)$ are given).

From executing this algorithm the following test program results:

Algorithm 3-7. Scanning test implementation

For all the values of $i = 1,2, \dots, n$
BEGIN
  Load the data registers $X(m^T)$ with D_i
  Carry out the tested working mode at the control values C
  Read the value of Y_i.
END.

Example 3-8

Generate a test program for testing the multiplier in *Figure 3-24*. In the DD we have two terminal nodes with the multiplier function. Let us choose the node $R_1 * R_2$ for testing. By activating the path to this node (shown in bold in *Figure 3-24*) we generate a control word $(y_2, y_3, y_4) = (0,3, 2)$. To find the proper values of R_1 and R_2 we need to descend to the lower level (e.g. gate level) and generate test patterns by a low level ATPG for the low level implementation of the multiplier. Let us have a test set of n test patterns $(D_{11},D_{21}; D_{12},D_{22}; \dots D_{1n},D_{2n})$ generated for the multiplier with inputs R_1 and R_2.

From above the following test program results:

For all the values of $i = 1,2, \dots, n$
BEGIN
  Load the data registers $R1 = D_{1i}$, $R2 = D_{2i}$
  Carry out the tested working mode at the control values $(y_2, y_3\ y_4) =$ (0,3,2)
  Read the value of Y_i.
END.

In the case when the control values are data dependent the algorithms become more complicated since the data found for nonterminal nodes by activating the paths in the DD should be consistent with the data found in processing of the terminal nodes.

Representing complex digital systems by a set of DDs. In general case a system is represented by a set of DDs, where each DD represents a part (subcircuit or component) of the system.

Consider a digital system consisting of the data and control parts as represented in *Figure 3-26*.

The data path of the system is partitioned into four subcircuits with functions described in *Table 3-5*. To model the system in the clock cycle basis it is reasonable to partition the system into components (subcircuits) in such a way that each subcircuit consists of a register with its input logic connected directly to other registers or to primary inputs. To simplify the hierarchical test generation, it is also reasonable to represent complex self-

contained blocks like adders, multipliers, ALUs, etc. as separate independent blocks. This allows during high-level test planning to reuse the local tests of these blocks if they are available

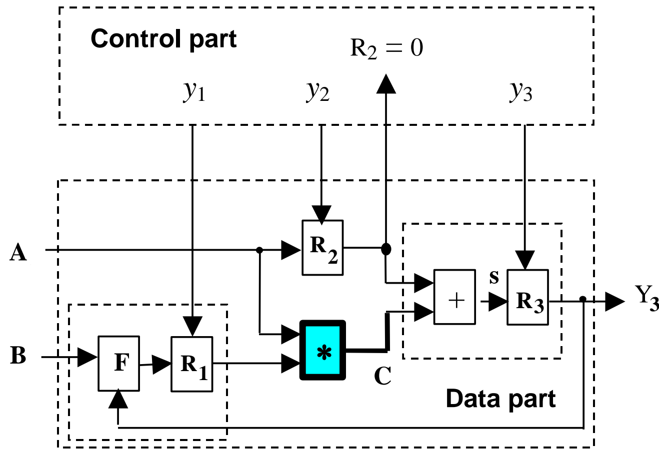


Figure 3-26. A system consisting of control and data parts

In the example system presented in *Figure 3-26*, y_1, y_2, and y_3 serve as control variables, *A* and *B* are the data inputs of the data path, R_1, R_2, and R_3 serve as data register variables, *C* is an output of the multiplier (input for the adder), and *Y* is the primary output of the data path. The functions of the subcircuits of the data path are presented also in *Table 3-5* (the previous state is denoted by apostrophe).

Table 3-5. Functions in components of the data-path in *Figure 3-26*

Block	Control	RTL operation	Function
R_1	$y_1 = 0$	$R_1 = 0$	Reset
	$y_1 = 1$	$R_1 = R'_1$	Hold
	$y_1 = 2$	$R_1 = F(B, R'_1)$	Special
R_2	$y_2 = 0$	$R_2 = 0$	Reset
	$y_2 = 1$	$R_2 = R'_2$	Hold
	$y_2 = 2$	$R_2 = A$	Load
	$y_2 = 3$	$R_2 = 2 * R'_2$	Shift
R_3	$y_3 = 0$	$R_3 = 0$	Reset
	$y_3 = 1$	$R_3 = R'_3$	Hold
	$y_3 = 2$	$R_3 = C + R'_2$	Add
C	None	$C = A * R'_1$	Multiply

The DD-model of the system is represented in *Figure 3-27*. The model consists of graphs G_{R2}, G_C, G_{R1} and $G_{Y,R3}$ for representing the functions of the register R_2, multiplier C and of two sub networks R_1 and $Y=R_3$, respectively, in the datapath surrounded by dotted lines in *Figure 3-26*.

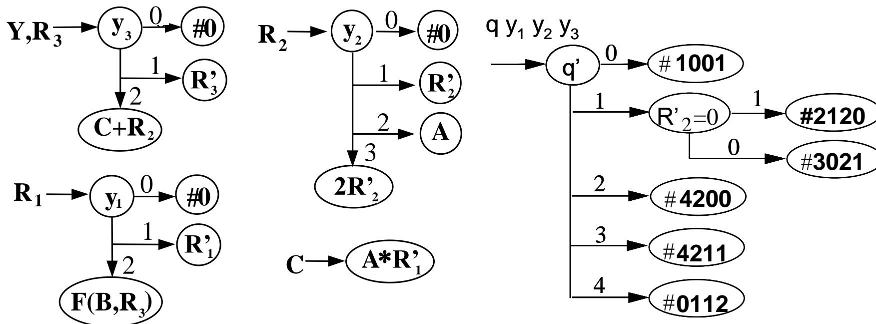


Figure 3-27. DD-representation of the digital system in *Figure 3-26*

The control part of the system is represented by the DD for calculating the value of the complex variable q,y_1,y_2,y_3 of the FSM given by the next state and output functions in *Table 3-6*. Here, q denotes the next state and q' denotes the current state variable. The variables y_1, y_2, and y_3 are the outputs of the FSM (the control inputs of the data path).

Table 3-6. FSM state transition and output table of the control-path in *Figure 3-26*

State	Condition	Nstate	Control signals		
q'		q	y_1	y_2	y_3
0		1	0	0	1
1	$R_2 = 0$	2	1	2	0
1	$R_2 \neq 0$	3	0	2	1
2		4	2	0	0
3		4	2	1	1
4		0	1	1	2

Hierarchical test generation for a complex digital system. According to the hierarchical approach, the tests are generated for all the nodes in all the DDs of the system description [50], [51]. For testing each node, the following steps of the procedure are carried out: high level deterministic **fault manifestation**, **fault propagation**, **constraints justification**, **constraints satisfaction** and low-level **random test generation**. The

procedure represents a systematic search and therefore an inconsistency in any stage will cause backtracking and returning to the last decision.

Fault manifestation in a system of DDs means generating a **symbolic local test** for a node in the given DD. The procedure is similar to generating conformity or scanning tests described above for the single DD model. A not yet tested node m in a given DD G_Y representing a function $Y = F(X)$, $X = (X_C, X_D)$, $X_D = (X_{D1}, X_{D2}, \ldots, X_{Dk})$ is chosen. Appropriate test (scanning or conformity type) is generated for testing the node m in G_Y. As the result a symbolic test vector is generated: $X_C = C$, $X_{D1} = D_1$, $X_{D2} = D_2, \ldots, X_{Dk} = D_k$, $Y = D$. Symbolic fault-effect value D is assigned to Y for fault propagating. The control values $X_C = C$, and the symbolic values (local test patterns) D_i, $i = 1,2, \ldots, k$ assigned to the data variables $X_{Di} \in X_D$ represent the constraints to be justified later. The constraints in the case of conformity test are represented in the form of inequality expressions.

Fault effect propagation. Subsequent to the manifestation phase, the fault effect has to be propagated to primary outputs of the whole circuit. The propagation procedure will follow, where decisions are made, both on the DDs of the data path as well as on the DD of the control part.

To propagate faults through the system network, for all the subcircuits as network components the lists of fault propagation modes like in *Table 3-4* should be created. For standard high-level components such lists can be presented as library information. For arbitrary high-level subcircuits the DD model can be used for direct representation of **fault propagation modes** [52]. In *Figure 3-28* a technique is illustrated, where the transparent fault propagation modes are directly inserted into the original DD-model in *Figure 3-27*. DDs allow representing this additional information more concisely than in the form of tables as shown in *Table 3-4*.

In the graph G_{R1}, an additional terminal node B is inserted. Activating the path in G_{R1} from the root node to B (by assigning y_1=2 and R'_3=0) is equivalent to creating an I-path (propagating a fault) from the input B to the register R_1 in *Figure 3-26*.

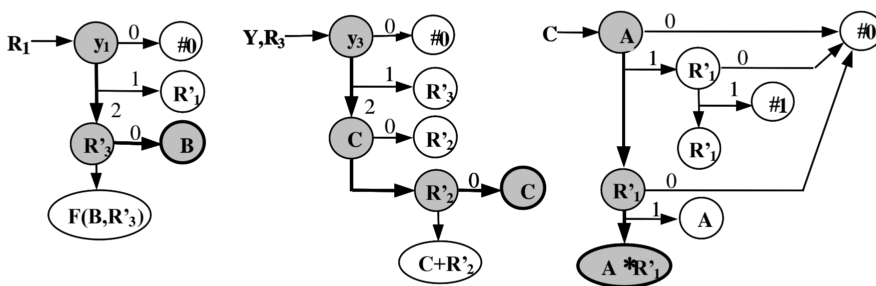


Figure 3-28. DDs with transparent fault propagation modes

In $G_{Y,R3}$, two additional terminal nodes with labels R'_2 and C are inserted. Activating the path from the root to the terminal node C (by assigning y_3=2 and R'_2=0 whereas the value of C remains free) is equivalent to creating an I-path (propagating a fault) from C to Y in *Figure 3-26*. Activating the path from the root to R'_2 (by assigning y_3=2 and C=0) is equivalent to propagating the fault from R'_2 to Y.

In the graph G_C, three additional terminal nodes with labels R'_1, A and constant 1 are inserted. Activating the path from the root to R'_1 (by assigning A=1) is equivalent to propagating transparently a fault from R'_1 to C in *Figure 3-26*. Activating the path from the root to A (by assigning R'_1=1) is equivalent to propagating the fault from A to C. Finally, by activating the path from the root to the terminal node with constant 1 (by assigning A=1 and R'_1=1) we solve the justification task C = 1.

Constraints justification. The aim of the justification is to backtrace the values and symbols that were set during the manifestation and propagation phases. The backtracing is performed via constraints extraction [50], [51]. Each time that a backward step is made during the justification, the contents of the constraints will be updated. Justification will end when all the variables in the constraints are primary inputs or constants.

The constraints are divided into two types: **path activation constraints** and **transformation constraints**. Path activation constraints are the constraints required to provide a transparent path through the circuit. Transformation constraints, in turn, reflect the value changes along the activated path. Path activation constraints are the constraints corresponding to the conditions to be satisfied in the FSM and to the values that are required to create a **transparent path** through a subcircuit of the system. Transformation constraints describe the changes in low-level test vectors on their way from primary inputs to the inputs of the module under test. During each time frame that is earlier than the manifestation time, the justification procedure selects a justification objective.

Constraint satisfaction. Subsequent to the constraint justification, the constraints have to be solved. In order to achieve that goal, any known Constraint Satisfaction Problem (CSP) solving algorithm can be applied. Only the path activation constraints are managed during constraint satisfaction. Transformation constraints are considered in the low-level test.

Low-level test generation. This step targets the gate-level structural faults in the current unit under test (UUT). During the low-level test, random values are generated to the variables of transformation constraints. The constraints are simulated to obtain the transformed vectors at UUT inputs, which, in turn, are applied to a fault simulation of the unit. If a fault is detected at the output of the module, it is also detected at the primary outputs

of the whole system. This is true because the fault effect propagation has previously guaranteed a transparent path from the output of the module to the primary outputs.

The vectors that detect previously undetected faults are compiled into final test vectors for the whole hierarchical circuit. This takes place by substituting the symbolic values in the high-level symbolic test frames by the actual values found during constraint satisfaction and low-level test.

To summarize the approach when generating a test for a module in a network, we have to propagate the fault effects from the output of the module up to the primary outputs of the network and also to propagate the test stimuli from the primary inputs of the network up to the inputs of the module. All these procedures are carried out on high-level DDs either by *Algorithm 3-4* or *Algorithm 3-6*.

Example 3-9

Consider the use of fault propagation modes embedded in DDs on an example of symbolic high-level test generation for a multiplier block *C* in *Figure 3-26*. Denote the set of local test patterns for the block *C* by *T*(*C*). To test the block *C* on high-level by assembling the patterns *T*(*C*), we have to create an I (or F)-path from the input *B* through the subnetwork R_1 to the input of *C* (for justifying the patterns *T*(*C*) on inputs of *C*), and to create a I(or F)-path from the output of *C* to the primary output *Y* for observing the test responses from *C*.

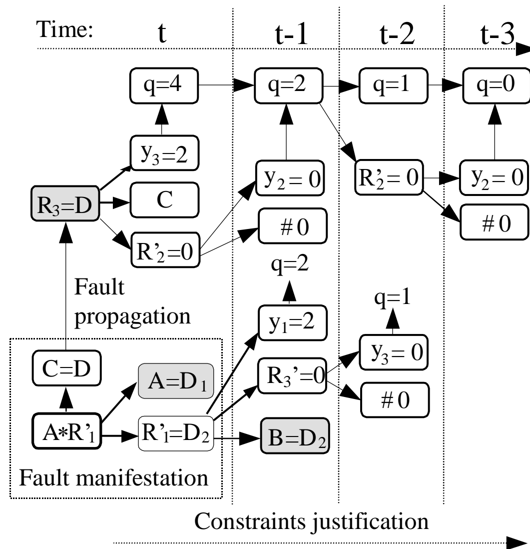


Figure 3-29. Test generation example for the block C in *Figure 3-26*

The test generation procedure with DDs in *Figure 3-27* and *Figure 3-28* is illustrated by the flow of activities in *Figure 3-29*. The generated final test sequence consisting of four symbolic test patterns is shown in *Table 3-7*, where t is the number of clock cycle.

Symbolic fault manifestation for the block $C = A * R'_1$ is carried out in the graph G_C in *Figure 3-28* by *Algorithm 3-6* of generating scanning test for the node $A * R'_1$. As the result, the highlighted path to the terminal node $A * R'_1$ is activated. The symbolic values are assigned to A and R'_1 ($A = D_1$, $R'_1 = D_2$) as justification objectives, and the symbolic fault-effect value D is assigned to C ($C = D$) for fault-effect propagation (see *Figure 3-29*).

Fault-effect propagation is produced in $G_{Y,R3}$ in *Figure 3-28* through the highlighted I-path which generates the constraints $R'_2 = 0$ and $y_3 = 2$ as the next justification objectives. To satisfy $y_3 = 2$, we find in the DD for the FSM in *Figure 3-27* a new justification objective $q = 4$. All these procedures assign values for variables at the current time frame t (the final test length is not yet known, we have to backtrace the time starting from the current moment denoted by t), see *Figure 3-29*.

Constraint justification task has now the following list: $A = D_1$, $R'_1 = D_2$, $R'_2 = 0$ and $q = 4$ (all for the current time period t). Since A is an input, the first task can be removed from the list.

To justify $R'_1 = D_2$, we have to go back to the previous time moment t-1 (the time shift is marked by the apostrophe at R'_1). To solve now the constraint $R_1 = D_2$, we generate in the graph G_{R1} an I-path through the nodes y_1, R'_3, and B. As a result, we find for t-1 the next justification tasks: $B = D_2$, $R'_3 = 0$, $y_1 = 2$. The first of these tasks, $B = D_2$, is already a solution, since B is the input.

To justify $R'_2 = 0$, we find in G_{R2} a new constraint $y_2 = 0$ for the time moment t-1.

To justify the state $q = 4$ in the DD of the vector $qy_1y_2y_3$ in *Figure 3-27* we can trace back to the states, either $q = 2$ or $q = 3$ at t-1. The constraint $y_1 = 2$ at t-1 in *Figure 3-29* can be solved also by two ways: $q = 2$ or $q = 3$ (see the same DD of $qy_1y_2y_3$ in *Figure 3-27*). On the other hand, $y_2 = 0$ at t-1 needs either $q = 0$ or $q = 2$. Intersection of these constraints gives the needed state $q = 2$ for the time period t-1.

Table 3-7. Symbolic test sequence for C in *Figure 3-26*

t	q	y_1 y_2 y_3	A	B	C	R_1	R_2	R_3	Y
1	0	0 **0** 1					**0**		
2	1	1 2 **0**						**0**	
3	2	**2** **0** 0		**D_2**		**D_2**	**0**		
4	4	1 1 **2**	**D_1**		**D**			**D**	**D**

The justification of the remaining constraints can be easily followed on the data dependency tree in *Figure 3-29*. The final test sequence consisting of four symbolic test patterns is given in *Table 3-7*. The symbolic values D_1 and D_2 can be substituted either by low-level generated deterministic local test patterns or by randomly generated local test patterns.

3.2.3 Test generation for microprocessors

Testing of microprocessors is a difficult problem because of the complexity and due to the lack of implementation details. The methods proposed in literature are based mainly on a functional level using the instruction set information [1], [2], [35], [44], [53], [54], [55].

Test generation with S-graphs. A microprocessor may be represented by interaction between registers according to an instruction set [35]. In [55] a model of **S-graph** describing interaction of registers for test generation purposes was proposed. The nodes of the graph represent registers. A pair of registers R_i and R_j is connected by an edge if there is an instruction involving these registers. The instructions and additional conditions which make the corresponding interaction active are shown on the edges. In addition to the register nodes two terminal nodes IN and OUT are added to represent the "outside world" of the microprocessor. They may be viewed as buses that connect the registers with a memory and peripheral devices.

In *Figure 3-30* a simple instruction set of a hypothetical microprocessor and its S-graph are shown.

I_1:	**MVI A,D**	**A = IN**	**I_6:**	**MOV A,M**	**A = IN**
I_2:	**MOV R,A**	**R = A**	**I_7:**	**ADD R**	**A = A + R**
I_3:	**MOV M,R**	**OUT = R**	**I_8:**	**ORA R**	**$A = A \vee R$**
I_4:	**MOV M,A**	**OUT = A**	**I_9:**	**ANA R**	**$A = A \wedge R$**
I_5:	**MOV R,M**	**R = IN**	**I_{10}:**	**CMA A,D**	**$A = \neg A$**

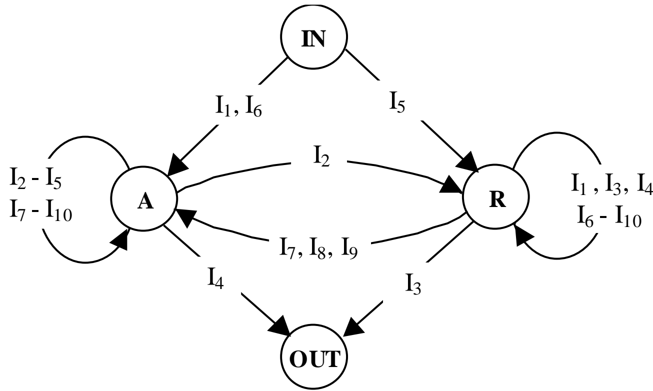


Figure 3-30. Instruction set of a microprocessor and its S-graph

Using the S-graph and the functional fault model of a microprocessor described in Chapter 2, the following test generation procedures have been developed [53], [55]:

- testing the **register-decoding** function (fault classes F1-F5),
- testing the **instruction-decoding** and **instruction-sequencing** function (fault classes F6-F8),
- testing the **data-storage** and **data-transfer** functions (F9-F13).

For testing the **data manipulation** functions no specific functional fault model has been proposed. The usual approach is to assume that tests for the data manipulation functions are developed by some other techniques [2]. In other words, it corresponds to the hierarchical approach where test planning is carried out on the functional (instructions set) level whereas test data for data manipulation units are generated on the gate-level.

The test of register-decoding functions involves writing and reading of registers. Whenever there exist several ways how to write or read a register, we select the shortest sequence. According to the start-small principle, registers are ordered for testing in increasing order of the lengths of read sequences.

The test of instruction-decoding and instruction-sequencing function is targeting the faults, which affect the execution of the instruction *I* and cause errors in the final results of the instruction (in the data transferred to the OUT node or in the register that can be read after *I* is executed). This should be true if **microinstructions** of *I* are not activated and/or if additional microinstructions are erroneously activated. Missing microinstructions are generally easy to detect. To detect the execution of additional parasitic microinstructions the **method of codewords** was proposed [53], [56].

Data-storage and data-transfer functions are tested together, because a test that detects stuck-at-faults on lines of a transfer path $A \rightarrow B$ also detects stuck-at-faults in the registers corresponding to the nodes *A* and *B*. A test for the transfer paths and for the registers is based on using different data patterns, so that

- every bit in a transfer path is set to both 0 and 1,
- every pair of bits is set to the values of 0 and 1 [55].

As an example, such test patterns for testing an 8-bit transfer path are presented in *Table 3-8* where the test patterns belong to rows, and the bits of the bus belong to columns.

Such a test detects all the stuck-at-faults on the lines of the transfer path and all the shorts between any pair of its lines.

Test generation with Decision Diagrams. The described S-graph based approach can be regarded as a special case of the test generation technique developed for DDs [28], [31] and described in Subsection 3.2.2. The first two procedures of testing register- and instruction decoding functions are equivalent to testing nonterminal nodes of DDs (conformity test), and the third procedure of testing data storage and data transfer functions together with testing of data manipulation faults are equivalent to testing of terminal nodes of DDs (scanning test).

Table 3-8. Test patterns for testing an 8-bit transfer path

	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1
2	1	1	1	1	0	0	0	0
3	1	1	0	0	1	1	0	0
4	1	0	1	0	1	0	1	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	1	1	1	1
7	0	0	1	1	0	0	1	1
8	0	1	0	1	0	1	0	1

Example 3-10

The DD-model of the instruction set in *Figure 3-30* consisting of three DDs G_{OUT}, G_A and G_R is shown in *Figure 3-31*.

The DD-model represents a conceivable network of three blocks in *Figure 3-32* with output variables *A*,*R*, and *OUT*, and with primary input variables: data variable *IN*, and control variable *I* which can have values from the set $\{I_1, I_2, \ldots I_{10}\}$.

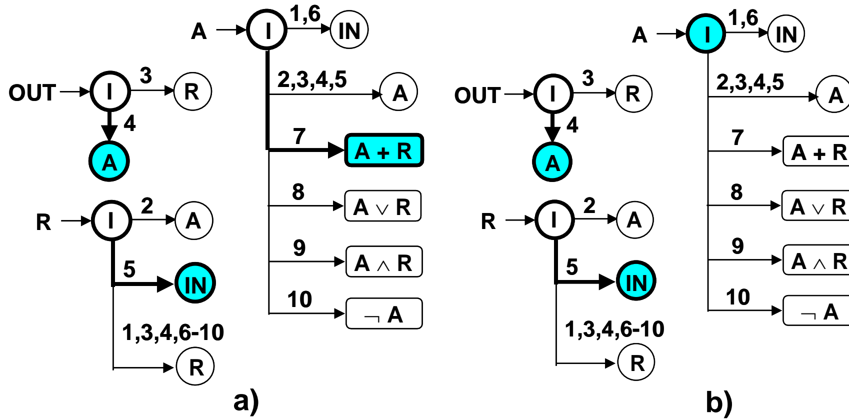


Figure 3-31. Decision diagrams of the microprocessor described in *Figure 3-30*

Transformation of the instruction set description into a structural model gives us a possibility to use standard procedures of fault manifestation, fault propagation and constraints justification in a similar way as it was considered in Subsection 3.2.2.

Consider (scanning) test generation according to *Algorithm 3-6* for the terminal node $A + R$ in G_A. The procedure is illustrated by highlighted activated paths in *Figure 3-31a,* by the fault manifestation, propagation and justification steps in *Figure 3-33a* and by the final test program created by *Algorithm 3-7* in *Figure 3-33b*.

In the fault manifestation step (test step in time frame t-1) a path is activated in G_A from the root node to the terminal node $A + R$, which gives $I(t\text{-}1) = I_7$ (see the bold lines in G_A in *Figure 3-31a*). The fault propagation step (observation step in the time frame t) to propagate the fault from A to OUT is carried out in G_{OUT}, which gives $I(t) = I_4$. The constraints justification step (load operations in time frames t-2, and t-3) is carried out in G_A by $I(t\text{-}2) = I_1$ for loading A with test data $IN(t\text{-}2)$ and in G_R by $I(t\text{-}3) = I_5$ for loading R with test data $IN(t\text{-}3)$. As the result of the test generation we get a symbolic test sequence: $I(t\text{-}3) = I_5$, $IN(t\text{-}3) = D_R$; $I(t\text{-}2) = I_1$, $IN(t\text{-}2) = D_A$; $I(t\text{-}1) = I_7$; $I(t) = I_4$.

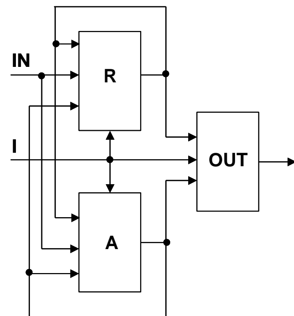


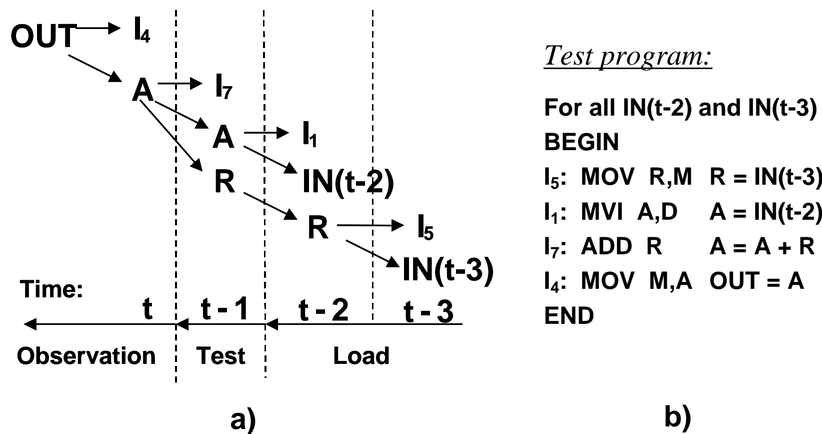
Figure 3-32. Structural model of microprocessor

The whole test generation procedure is presented in *Figure 3-33a*, and the resulting test program according to Algorithm 3-7 is given in *Figure 3-33b*. The test data $\{D_R, D_A\}$ for testing the adder $A + R$ are generated at the low level by using gate-level specification of the adder.

Consider now (conformity) test generation according to *Algorithm 3-4* for the nonterminal node I in G_A, which is illustrated in *Figure 3-31b* and *Table 3-9*.

In the fault manifestation step (time frame t-1) a local test pattern is generated in G_A for testing the node I. Since I is the root node, and all its successors are the terminal nodes, no paths should be generated in G_A. The local test pattern consists of the symbolic value $D \in \{I_1, I_2, \ldots I_{10}\}$ assigned to the control variable $I(t\text{-}1)$, and of test data $IN(t\text{-}1)$, $A(t\text{-}1)$ and $R(t\text{-}1)$ found as a solution of the inequality $IN \neq A \neq A{+}R \neq A{\vee}R \neq A{\wedge}R \neq \neg A$. The values of $A(t\text{-}1)$ and $R(t\text{-}1)$ need to be justified as constraints. Since IN is an input variable, $IN(t\text{-}1)$ does not need justification.

The fault propagation and constraints justification steps are similar to the previous example of testing $A + R$.



Test program:

For all IN(t-2) and IN(t-3)
BEGIN
I_5: MOV R,M R = IN(t-3)
I_1: MVI A,D A = IN(t-2)
I_7: ADD R A = A + R
I_4: MOV M,A OUT = A
END

b)

Figure 3-33. Test program generation for a microprocessor

To propagate the fault (in the time frame t) from A to OUT we assign $I(t) = I_4$ in G_{OUT}. To justify the constraint $A(t\text{-}1)$ in the time frame t-2 we activate by $I(t\text{-}2) = I_1$ in G_A the path from the root to the terminal node IN, and assign

$IN(t\text{-}2) = A(t\text{-}1)$. To justify $R(t\text{-}1)$ at t-3 we activate by $I(t\text{-}3) = I_5$ in G_R the path from the root to *IN*, which gives $IN(t\text{-}3) = R(t\text{-}1)$.

Table 3-9. Test data generation for testing the node I in G_A

Data	**IN**		**0**
	A		**101**
	R		**110**
Functions	I_1, I_6	**IN**	**0**
	$I_2, I_3\ I_4, I_5$	**A**	**101**
	I_7	**A + R**	**1011**
	I_8	$A \vee R$	**111**
	I_9	$A \wedge R$	**0**
	I_{10}	$\neg A$	**0**

Test data generation for solving the inequality $IN \neq A \neq A{+}R \neq A{\vee}R \neq A{\wedge}R \neq \neg A$ is illustrated in *Table 3-9*. The inequalities are solved by choosing bit by bit the proper data for $IN(t\text{-}1)$, $A(t\text{-}1)$, and $R(t\text{-}1)$ to make the values of all the expressions $IN, A, A{+}R, A{\vee}R, A{\wedge}R, \neg A$ different.

The resulting test program according to *Algorithm 3-5* can be constructed as follows:

For all $D = \{I_1, I_2, \ldots I_{10}\}$
BEGIN
$I(t\text{-}3) = I_5, IN(t\text{-}3)=110$;
$I(t\text{-}2)= I_1$, $IN(t\text{-}2)=101$;
$I(t\text{-}1)= D$, $IN(t\text{-}1)=0$;
$I(t)= I_4$.
END

In the discussion above we have described the general ideas of testing microprocessors based on generic architectures. An overview of test strategies developed for real life microprocessors may be found in [35].

3.3 FAULT SIMULATION

Fault simulation plays an important role in the ATPG process. A fault simulator has to classify the given target faults in DUT as detected or undetected by given test stimuli (patterns). Fault simulation does not try to generate new test patterns, but determines the fault coverage of existing vectors. The basic fault simulation techniques are serial, parallel, deductive and concurrent. A special method is the critical path tracing technique where test patterns are generated together with determination of detected faults they cover during one path tracing through the DUT structure from primary outputs toward primary inputs [1], [2]. The **serial fault simulation** is the

simplest algorithm for simulating faults injected into the DUT structure by modifying the circuit description for a target fault and using a true-value simulator.

3.3.1 Parallel fault simulation method

The **parallel fault simulation** algorithm combines two separate concepts – single-fault propagation and parallel-pattern evaluation and was widely used in the 1960s and 70s. It was implemented in various commercial ATPG systems, e.g. the HILO and TEGAS simulators. The idea of the parallel fault simulation is to use the bit-parallelism of logic operations in the computer. For example in a 32-bit computer word, an integer consists of a 32-bit binary vector. This allows a simultaneous simulation of 32 copies of circuit (1 fault-free circuit and 31 faulty copies) with identical connection, but with possibly different signal values [1], [2]. The faults detection is reported by different values in bit positions in comparison with the fault-free outputs to input stimuli. The automatic parallel fault simulation uses the following expressions:

- A fault is represented by a mask on a signal line (index i is a bit position in masks):

$$mask\ (s)_i = 1 \quad \text{- if a fault on the line } s \text{ exist,} \qquad (3\text{-}1)$$
$$mask\ (s)_i = 0 \quad \text{- if a fault on the line } s \text{ does not exist.}$$

- SAF0 or SAF1 are represented by *p_value* defined for the signal line:

$$p_value\ (s)_i = 1 \quad \text{- if the fault on } s \text{ is SAF1}$$
$$p_value\ (s)_i = 0 \quad \text{- if the fault on } s \text{ is SAF0.} \qquad (3\text{-}2)$$

The new value on the signal line s is calculated according to the following expression:

$$s' = s \,.\, \overline{mask\ (s)} + mask\ (s) \,.\, p_value\ (s), \qquad (3\text{-}3)$$

where operation "." and "+" is logical multiplication and addition.

A simple example of the parallel fault simulation is shown on a circuit and input pattern (ABCD) = (1101) presented in *Figure 3-34*.

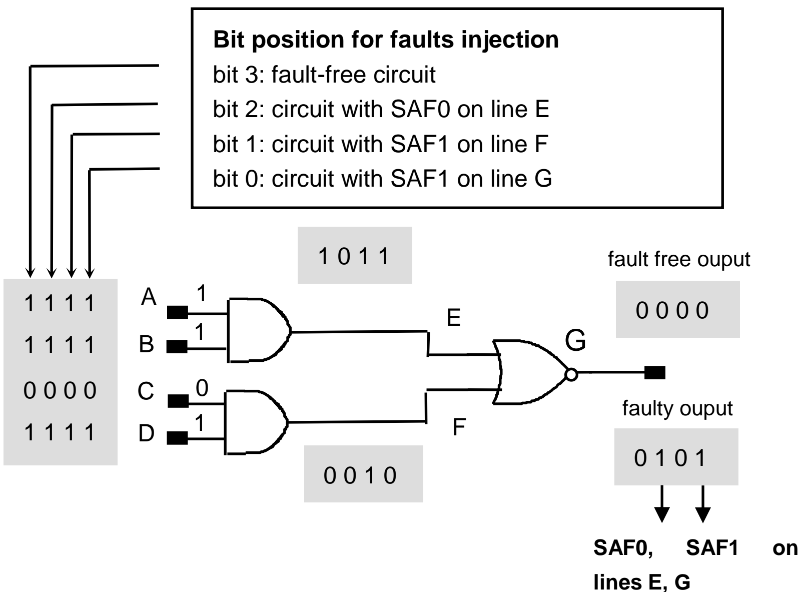


Figure 3-34. An example of parallel fault simulation

<u>*Example 3-11*</u>:

Consider circuit c17 in *Figure 3-15* and 8 faults selected for parallel fault simulation described in *Table 3-10*. The masking values and corresponding *p_values* for the selected faults are shown in *Table 3-11*. The input vector for c17 is (ABCDE) = (01110). Find all faults covered by the input pattern.

Table 3-10. A list of faults for c17

Bit position	Faults	signal line
1	fault-free	
2	A/SAF0	A
3	A/SAF1	A
4	B/SAF0	B
5	B/SAF1	B
6	C/SAF0	C
7	D/SAF1	D
8	C1/SAF1	Fan from C
9	C2/SAF1	Fan form C

Table 3-11. Data for parallel simulation

Signal line	*mask*	*p_value*
A	011000000	001000000
B	000110000	000010000
C	000001000	000000000
D	000000100	000000100
C1	000000010	000000010
C2	000000001	000000001

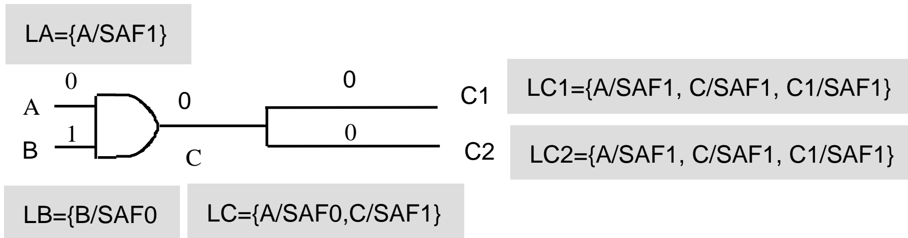
The steps of the parallel fault simulation are:

1. Fault-free simulation with the input patterns: A = (000000000), B = (000000000), C = (111111111), D = (000000000), E = (000000000); Outputs are Y1 = (000000000), Y2 = (0000000000).
2. The new value for signal line A is calculated according (3-1):
   A' = A . $\overline{mask}$ (A) + *mask* (A) . *p_value* (A)
   A' = (000000000).(100111111)+(011000000).(001000000)
   A' = (001000000).
3. A similar computation is done for other signal lines B, C, D:
   B' = (000010000), C' = (111110111), D' = (000000000).
4. The logical operations are computed with the new masked values, e.g.:
   NOT(A'.C') = (110111111) and NOT(C'.D') = (111111111).
   Then the new values are calculated according to steps 2 and 4 based on the circuit structure. If any fault is modelled on signal line S, then S=S'.

5. The values on outputs are Y1= (001010000) and Y2=(000010000) and according to *Table 3-11* SAF1 on line A and SAF1 on B are covered by pattern (ABCDE) = (01110).

3.3.2 Deductive and concurrent fault simulation techniques

In the **deductive fault simulation** technique, only the fault-free DUT is simulated. All signal values in a faulty circuit are deducted from the fault-free circuit values and the circuit structure. Since the circuit structure is the same for all faulty circuits, all deductions are carried out simultaneously. Thus, a deductive fault simulator finds all faults in a single pass of the fault-free simulation augmented with the deductive procedures for creating and



propagating fault lists from primary inputs towards primary outputs. In this process, fault lists are generated for each signal.

Figure 3-35. Fault list creation and propagation through gate AND

An example of the fault list propagation through basic gate AND is shown in *Figure 3-35.* LA, LB, LC, LC1, LC2 are lists of faults detected on signal lines A, B, C, C1and C2. They are propagated through the DUT structure from primary inputs to primary outputs using the rules reported in *Table 3-12*. I_0 and I_1 mean the numbers of logic 0 and logic 1, respectively, set up on gate's inputs.

Table 3-12. Rules for fault list propagation through basic gates

Gate	No. of logical values 0 or 1	Rules
AND/NAND	$I_0 = 0$	$\bigcup_{i \in I_1} LX_i$
	$I_1 \neq 0$	$\bigcap_{i \in I_0} LX_i - \bigcup_{i\, j \in I_1} LX_i$
OR/NOR	$I_1 = 0$	$\bigcup_{i\, j \in I0} LX_i$
	$I_0 \neq 0$	$\bigcap LX_i - \bigcup LX_i$

Gate	No. of logical values 0 or 1	Rules		
			$i \in I_1$	$i \in I_0$
NOT			LX_i	

Example 3-12:

Consider fault list propagation through gate AND with 3 inputs using the test vector (ABC) = (101); the assumption that the next fault lists were propagated to lines A, B, C (see *Figure 3-36*): LA = {P/t1, Q/t0, R/t1, **A/t0**}, LB = {Q/t0, R/t0, S/t1, **B/t1**} LC = {R/t1, S/t0, **C/t0**}, where P, Q, R, S are lines before A, B, C, D. The list of faults is created for the output of AND (E) according to the expression: LE = (LB – LA ∪ LC) ∪ E/t1 and the resulting list of faults on line E is LE = {R/t0, S/t1, B/t1, E/t1}.

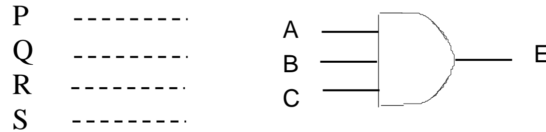


Figure 3-36. Fault lists propagation in c17 to one output

Example 3-13:

Find all faults in circuit c17 covered by vector (ABCDE) = (01110) observable on output Y1 using the deductive fault simulation. The results are shown in *Figure 3-37* based on the following steps:

Step 1: Fault-free simulation for the input test.

Step 2: Fault lists creation on the primary inputs.

Step 3: Fault lists propagation using the rules defined in *Table 3-12* up to primary outputs.

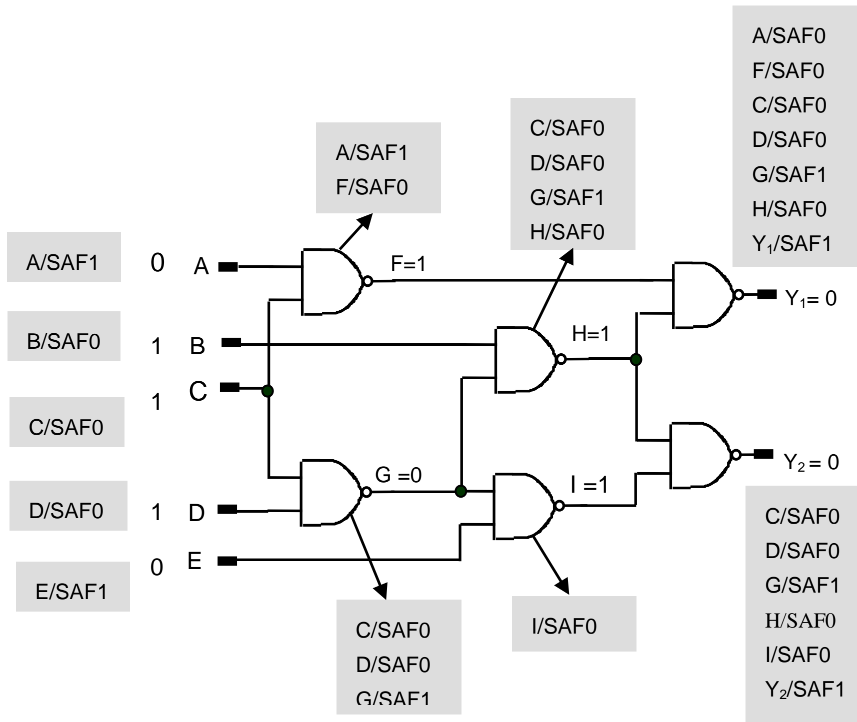
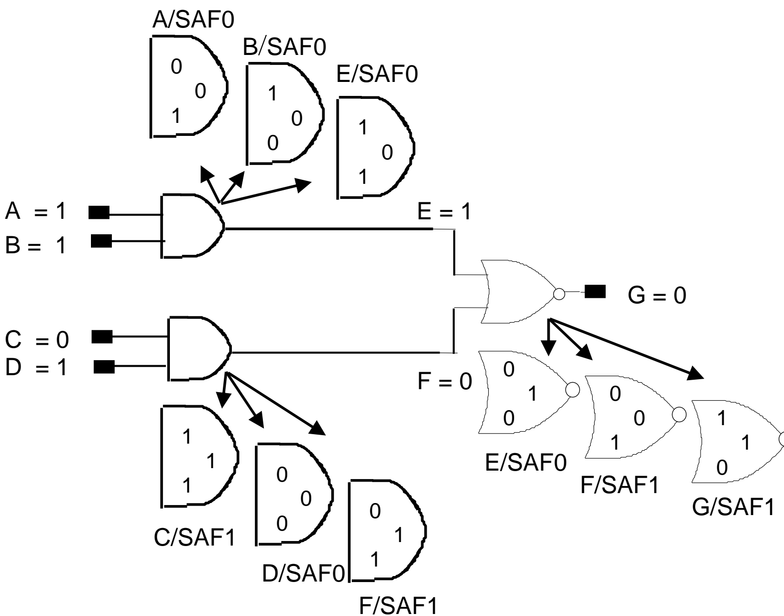


Figure 3-37. Fault lists propagation in c17 to one output

The disadvantage of the deductive fault simulation is in creation of fault lists for each test. All generated fault lists for the applied input vector are deleted and they have to be created again for the new input vector. Therefore the deductive fault simulation was modified, and a new concurrent fault simulation technique has been developed. Another problem has to be solved if DUT contains a feedback where the fault list of a signal may change several times. Only after the fault lists become stable, the simulator proceeds with the next vector.

3.3.3 Concurrent fault simulation techniques

The **concurrent fault simulation** technique is the most general fault



simulation technique. It can handle various circuit models, faults, signal states and timing models. It basically extends the event-driven simulation method to the simulation of faults in the most efficient way. Beside the fault type, a fault list created for each cell in DUT contains also faulty values on the inputs and output of each cell. An example is shown in *Figure 3-38* where all possible faulty states of each element are described.

Figure 3-38. Fault list propagation using the concurrent fault simulation

While parallel and deductive simulation techniques are reasonably efficient for two logical values, using three and more logical values significantly increases the computational requirements, and the techniques are not practical in this case. Both methods are only partially compatible with functional-level modelling, as they can process only components that can be entirely described by Boolean equations. Both parallel and deductive simulation, the basic data structures and algorithms are strongly dependent on the number of logical values used in modelling. By contrast, the concurrent method provides only a mechanism to represent and maintain the differences between the good circuit and a set of faulty circuits, and this mechanism is independent of the way the circuits are simulated. It is totally compatible with functional-level modelling and can support mixed-level and hierarchical modelling. The main disadvantage is that it requires more memory than other methods, but in comparison with the deductive fault simulation the concurrent method is faster and suitable for increasingly complex circuits and evolving technology.

3.3.4 Critical path tracing method

The critical path tracing technique is a fault independent method. For every input vector, critical path tracing first simulates the fault-free circuit then it determines the detected faults by ascertaining which signal values are critical. The technique is based on the next two definitions and Lemma [2].

Definition 3.2: A signal line l has a critical value v in the test vector t if t detects the fault SAF_v. A line with a critical value in t is said to be critical in t.

Definition 3.3: A gate input is sensitive (in a test t) if complementing its value changes the value of the gate outputs.

Lemma 3.1: If a gate output is critical, then its sensitive inputs, if any, are also critical.

The primary outputs are immediately critical in any test. Starting at the primary outputs the other critical lines are found by tracing towards primary inputs of DUT. This process determines the paths composed of critical lines, called critical paths. It uses the concept of sensitive inputs. The sensitive inputs of a gate can be easily identified during the fault-free simulation of DUT, as scanning for inputs with controlling value is an inherent part of gate evaluation. The sensitive inputs of a gate with two or more inputs are easily determined as follows:

1. If only one input j has the controlling value of the gate, then j is sensitive.
2. If all inputs have controlling value, then all inputs are sensitive.
3. Otherwise no input is sensitive.

If all critical lines are found in DUT, and values were assigned to primary inputs then all faults on critical lines are covered by the received test pattern. An example is shown in *Figure 3-39* (critical lines are bold).

Comparing with conventional fault simulation, the features of critical path tracing are as follows:

- It directly identifies the faults detected by a test without simulating the set of all possible faults.
- It deals with faults only implicitly.
- It is based on a path tracing that does not require computing values in the faulty circuits by gate evaluations or fault list processing.
- It is an approximate method.

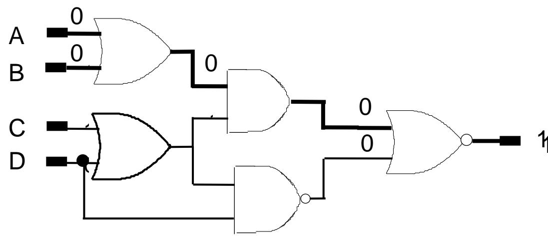


Figure 3-39. Critical lines in a circuit

The critical path tracing method is faster and requires less memory than the conventional fault simulation techniques. Some experimental results have shown that the critical path tracing technique is faster than the concurrent fault simulation but this method has to encounter some problems in circuits with reconvergent fanouts. If not all lines from one root are critical the root does not have to be critical. The critical path tracing technique can produce conflicts, self-masking, multiple-path sensitisation and overlaping among primary output cones.

3.3.5 Macro-level fault simulation

In gate-level deductive fault analysis we need different fault list propagation formula to be stored in the data base for each gate and for each input pattern. This makes impossible to carry out the fault propagation procedure at higher than gate levels e.g. at **macro levels** where the macros may represent arbitrary **Boolean functions**.

Using **Boolean full differentials** it would be possible to generalize the gate-level deductive fault analysis approach to higher macro levels.

Consider a macro (subcircuit) with a Boolean function $y = F(X) = F(x_1, x_2, \ldots x_n)$. Introduce a **Boolean differential** dx_i so that $dx_i = 1$ when the value of the variable x_i is changing because of a fault, and $dx_i = 0$ otherwise. In a similar way, a differential dy can be introduced for the output variable y.

The full Boolean differential [58] dy of the output variable y describes the **cause-effect relationship** between any value changes of input variables $x_i \in X$ and the output variable y. The expression of the full differential dy can be presented also in the vector form where each component of the vectors $dx_i = (dx_{i,1}, dx_{i,2}, dx_{i,m})$ and $dy = (dy_1, dy_2, dy_m)$ represents the behavior of the corresponding variables x_i and y for different possible faults $F_1, F_2, \ldots, F_m$. Representing the **input fault lists** $L_{x1}, L_{x2}, \ldots, L_{xn}$, in a vector form we can easily calculate the **output fault list** L_y in a vector form.

<u>*Example 3-14*</u>

Consider an example of the gate-level deductive fault simulation shown in *Figure 3-40*.

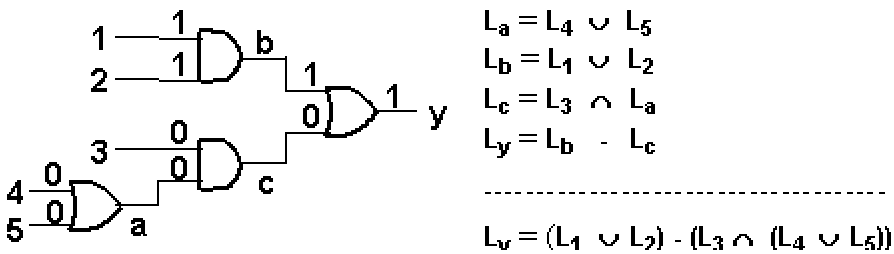


Figure 3-40. Gate-level deductive fault simulation

The Boolean full differential of the circuit is:

$$dy = y \oplus (x_1 \oplus dx_1)(x_2 \oplus dx_2) \vee [(x_3 \oplus dx_3)((x_4 \oplus dx_4) \vee (x_5 \oplus dx_5))]$$

Substituting the variables x_i by their values at the given test pattern we get

$$dy = (dx_1 \vee dx_2)\overline{dx_3(dx_4 \vee dx_5)}$$

Let us have the following fault lists at inputs: L_1 = {1,3,4}, L_2 = {2,3,4}, L_3 = {2,4}, L_4 = {1,2,5}, L_5 = {3,4}. By gate level deductive fault list calculation we get L_a = {1,2,3,4,5}, L_b = {1,2,3,4}, L_c = {2,4}, L_y = {1,3}.

In the vector form we have: dx_1 = 10110, dx_2 = 01110, dx_3 = 01010, dx_4 = 11001, dx_5 = 00110. Calculating the full Boolean differential we get dy = 10100 which corresponds to the fault list L_y = {1,3}.

3.3.6 Hierarchical fault simulation

Gate-level fault simulation methods have proved to be very time-consuming for complex digital systems. Hierarchical methods allow taking the advantage of high level information while simulating tests for gate-level faults.

In the hierarchical approach, the fault analysis for the given test is carried out for the blocks of the higher level network block by block. At each iteration, a target block is chosen and represented on the gate level whereas all other blocks are represented on the RT level. Always when the target block is simulated, the faults are determined which cause erroneous output behaviour of the block at the given input pattern and given state of the target block. The propagation of detected faults of the target block is analyzed by using the RTL or other high-level description.

An example of the approach is illustrated in

Figure 3-41 where the network of the system consists of three blocks: A, B, and C. The block B is taken here as the target block for the fault analysis, and is represented on the gate level. Test sequence is simulated for the whole system on the higher level, pattern by pattern. When the target block B is reached by the first input pattern P, low level fault analysis in B is carried out, and the subset of all faults R activated in B by the pattern P at the given state of B is calculated. For each fault $r \in R$, the corresponding faulty output pattern $P(r)$ of the block B is calculated. Activated faults are grouped into subsets $R_i \subseteq R$, so that for each faulr $r \in R_i$ the output pattern $P(r) = P_i$ would be the same. The fault-free pattern P, and all the faulty patterns P_1, …,P_k are simulated through other blocks of the network on the higher level. If $P_i \neq P$ at the observable output then the faults R_i are claimed as detected. The detected faults can be removed from further analysis (called **fault dropping** [2]). In general, at the output of each higher level block, a data structure (**complex test pattern**) $D = \{P, (P_1,R_1), \ldots, (P_k,R_k)\}$ will be generated where $R_1 \cup R_2 \cup \ldots \cup R_k \subseteq$ R. When the target block B (UUT) is reached by a complex pattern D, all the patterns in $\{P,P_1, \ldots,P_k\}$ should be

fault simulated on the low-level. For P, new faults will be determined which are activated by P, and for all the faults in each R_i it will be checked if they are propagated by the pattern P_i again through B or not. After that, a new data structure D will be created for the output of B.

Consider a block with a function of n arguments $y = f(x_1, \dots x_n)$ (x_i is either input or a state variable), and a set of complex patterns $D_i = \{P_{i,0}, (P_{i,1},R_{i,1}), \dots, (P_{i,ki},R_{i,ki})\}$, $i = 1,2, \dots, n$ where $P_{i,0}$ is the fault-free pattern on the input x_i, and $P_{i,j}$ are the possible faulty patterns on the same input for the cases of faults in $R_{i,j}$, j=1,2,… ki. Denote by

$$R_i = R_{i,1} \cup R_{i,2} \cup \ldots \cup R_{i,ki} \subseteq R$$

the set of all faults propagated to the input x_i, where R is the whole set of faults in the system to be simulated.

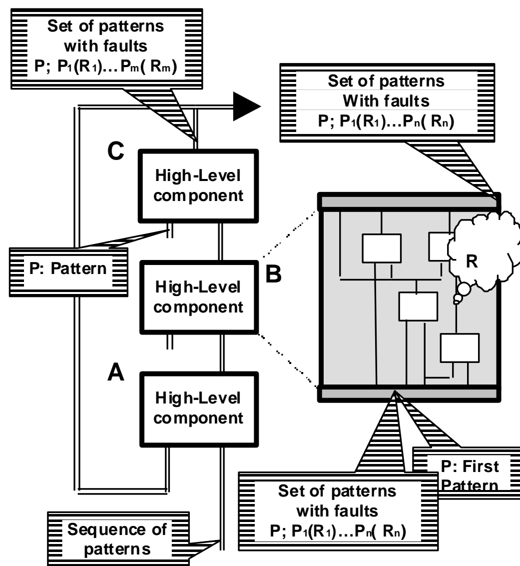


Figure 3-41. Multi-level fault simulation

The multi-level fault simulation of digital systems is carried out by the following procedures.

Algorithm 3-8. High-level fault simulation

To fault simulate a high-level block for a complex pattern $D = (D_1,\dots, D_n)$, first, the correct behaviour of the block for the pattern $(P_{1,0},\dots,P_{n,0})$ is calculated on the high-level. Then, for all the faults

$$r \in R' = \bigcup_{i=1,n} (\bigcup_{j=1,ki} R_{ij}) \subseteq R$$

propagated to the block, all the possible combinations of patterns $(P_1(r),\ldots,P_n(r))$ where for each $i = 1, \ldots, n$ either $P_i(r) = P_{i,0}$ if $r \notin R_i$, or $P_i(r)= P_{ij}$ if $r \in R_{ij}$, are simulated on high-level.

Algorithm 3-9. Low-level fault simulation

To fault simulate the target low-level block for a complex pattern $D = (D_1,\ldots, D_n)$, first, the correct behaviour of the block for the pattern $(P_1,\ldots,P_n)$ is calculated, and the faults causing erroneous output behaviour of the block at this pattern are determined. Then, for all the faults $r \in R'$ in the complex pattern D which have propagated back to the same target block, all the possible combinations of patterns $(P_1(r), \ldots,P_n(r))$ are simulated in the presence of the given fault r.

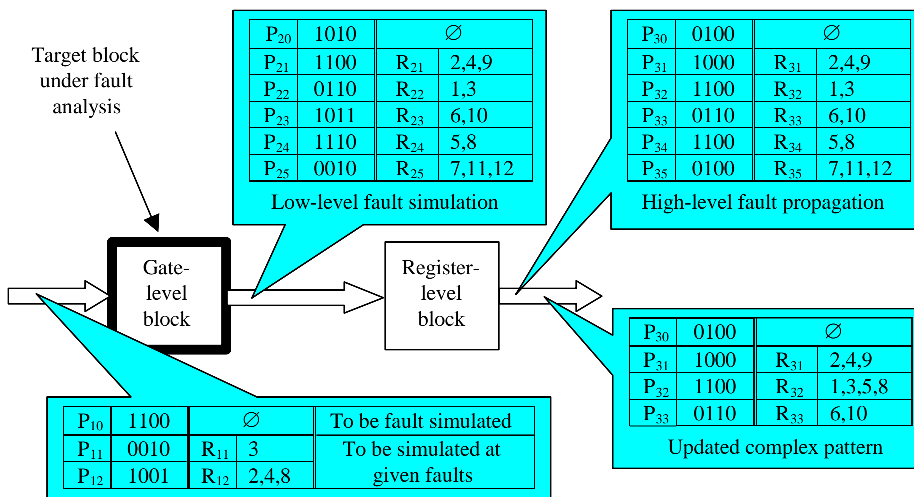


Figure 3-42. Fault propagation in the multi-level fault simulation

Example 3-15

An example of the fault propagation in multi-level fault simulation is presented in *Figure 3-42*. Assume the input 1 of the target block represented on gate-level is reached by a complex test pattern $D_1 = \{P_{10}, (P_{11},R_{11}), (P_{12},R_{12})\} = \{1100, 0010(3), 1001(2,4,8)\}$. The faults 2,3,4,8 of the block have been propagated through the simulated feedback loop back to the same block. By the fault analysis we find that 12 faults are detected by the current input pattern $P_{10} = 1100$ (or propagated by faulty patterns 0010 and 1001) on the output of the target block. They cause 5 different output patterns which differ from the expected one 1010. All the 6 output patterns are simulated now on high-level for the next block. From $P_{30}=P_{35}$ we conclude that the

faults 7,11 and 12 are self-masked – the pair (P_{35},R_{35}) is removed from further simulation. From P_{32}=P_{34} we conclude that the faults 1,3 and 5,8 cannot be distinguished, and we include them into the same group of faults R_{32} = {1,3,5,8}.

For fault analysis of the target block on the lower level and fault propagation through other blocks on the higher level we can use the same mathematical model of decision diagrams discussed in this chapter. To increase the speed of low-level fault analysis, we can use macro networks instead of gate networks where each macro is represented by a SSBDD. For fault propagation through other blocks we can use word-level DDs [57].

3.4 FAULT DIAGNOSIS AND FAULT LOCALISATION

A unit under test (UUT) fails when its **observed behavior** is different from its **expected behavior**. The task of the fault diagnosis is to locate the fault(s) in a structural model of the UUT. The degree of the accuracy to which faults can be located is called **diagnostic resolution**. **Functionally equivalent faults** (FEF) cannot be distinguished. 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-localization test** [2].

The fault diagnosis process is often hierarchical, carried out as a top-down process (with a system operating in the field) or bottom-up process (during the fabrication of the system).

In the **top-down approach** (system → boards → ICs) first-level diagnosis may deal with "large" **replaceable parts** of a system like boards called also **field-replaceable units**. The faulty board is then tested in a **maintenance center** to locate the faulty component (IC) on the board. Accurate location of faults inside a faulty IC may be also useful for improving its manufacturing process.

In the **bottom-up approach** (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.

The **rule of 10**: if it costs $1 to test an IC, the cost of locating the same defective IC when mounted on a board and repairing the board is about $10; when the defective board is plugged into a system, the cost of finding the fault and repairing the system is $100.

In manufacturing, the most likely faults are **fabrication errors** affecting the interconnections between components; in the field the most likely faults

are **physical failures** internal to components (because every UUT has been successfully tested in the past). Knowing the most likely class of faults helps in fault location.

3.4.1 Combinational fault diagnosis methods

Most fault diagnosis methods are based on using **fault tables** or **fault dictionaries**, which can be created by fault simulation. To locate faults, one tries to match the actual results of test experiments with one of the precomputed expected results stored in the database (fault table or dictionary). The result of the **test experiment** represents a combination of effects of the fault to each test pattern. That's why we call this approach **combinational fault diagnosis** method. If this look-up process is successful, the fault table (dictionary) indicates the corresponding fault(s).

In general, a fault table is a matrix FT = $|a_{ij}|$ where columns F_j represent faults, rows T_i represent test patterns, and $a_{ij} = 1$ if the test pattern T_i detects the fault F_j, otherwise if the test pattern T_i does not detect the fault F_j, $a_{ij} = 0$.

Denote the actual result of a given test pattern by 1 if it differs from the precomputed expected one, otherwise denote it by 0. The **result of a test experiment** is represented by a vector $E = |e_i|$ where $e_i = 1$ if the actual result of the test pattern does not match with the expected result, otherwise $e_i = 0$. Each column vector f_j corresponding to a fault F_j represents the result of the test experiment in the case when the fault F_j is present.

Three cases are now possible depending on the quality of the test patterns used for carrying out the test experiment and on the thoroughness of the fault set taken into account:

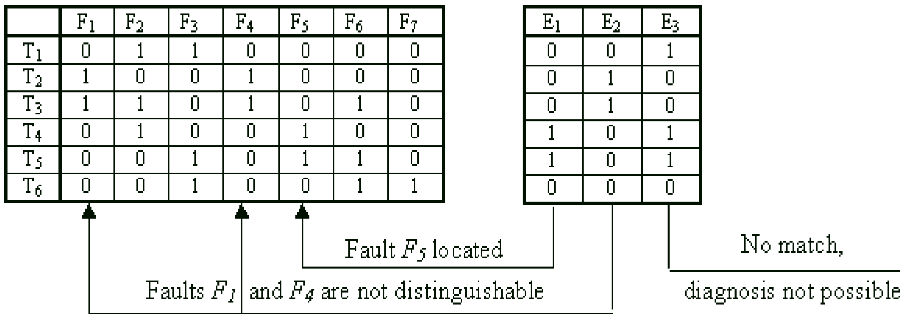
1. The test result E matches with a single column vector f_j in *FT*. This result corresponds to the case where a single fault F_j has been located. In other words, the maximum diagnostic resolution has been obtained.

2. The test result E matches with a subset of column vectors $\{f_i, f_j \dots f_k\}$ in FT. This result corresponds to the case where a subset of **indistinguishable faults** $\{F_i, F_j \dots F_k\}$ has been located. To distinguish these faults additional test patterns are needed.

3. No match for E with column vectors in FT is obtained. This result corresponds to the case where the given set of vectors does not allow carrying out fault diagnosis. The set of faults described in the fault table must be incomplete (in other words, the real existing fault is missing in the fault list considered in FT).

In the example in *Figure 3-43* the results of three test experiments E_1, E_2, E_3 are demonstrated. E_1 corresponds to the first case where a single fault is located, E_2 corresponds to the second case where a subset of two indistinguishable faults is located, and E_3 corresponds to the third case where

no fault can be located because of the mismatch of E_3 with the column vectors in the fault table.



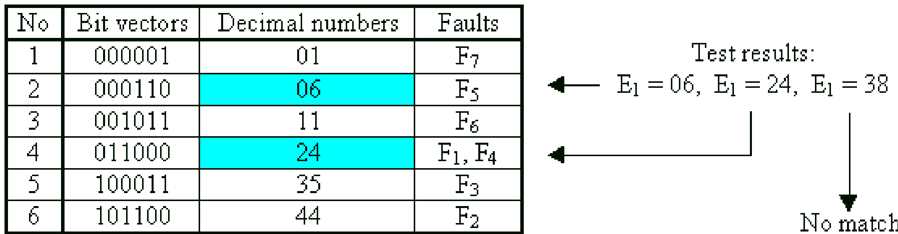
	F_1	F_2	F_3	F_4	F_5	F_6	F_7
T_1	0	1	1	0	0	0	0
T_2	1	0	0	1	0	0	0
T_3	1	1	0	1	0	1	0
T_4	0	1	0	0	1	0	0
T_5	0	0	1	0	1	1	0
T_6	0	0	1	0	0	1	1

E_1	E_2	E_3
0	0	1
0	1	0
0	1	0
1	0	1
1	0	1
0	0	0

Figure 3-43. Fault diagnosis with the fault table

Fault dictionaries (FD) contain the same data as the fault tables with the difference that the data are reordered. In FD 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 (see the example) or by some kind of compressed signature.

An example of the fault dictionary for the fault table in *Figure 3-43* is shown in *Figure 3-44*.



No	Bit vectors	Decimal numbers	Faults
1	000001	01	F_7
2	000110	06	F_5
3	001011	11	F_6
4	011000	24	F_1, F_4
5	100011	35	F_3
6	101100	44	F_2

Figure 3-44. Fault dictionary

To reduce large computational effort involved in building a fault dictionary, the detected faults are dropped from the set of simulated faults during fault simulation. Hence, all the faults detected for the first time by the same vector will produce the same column vector (**signature**) in the fault table, and will be included in the same **equivalence class of faults**. In this case the test experiment can stop after the first failing test, because the information provided by the following tests is not used. Such a test experiment achieves a lower diagnostic resolution. A tradeoff between

computing time and diagnostic resolution can be achieved by dropping faults after k >1 detections.

	F_1	F_2	F_3	F_4	F_5	F_6	F_7
T_1	**0**	**1**	**1**	**0**	**0**	**0**	**0**
T_2	**1**	0	0	**1**	**0**	**0**	**0**
T_3	0	0	0	0	**0**	**1**	**0**
T_4	0	0	0	0	**1**	0	**0**
T_5	0	0	0	0	0	0	**0**
T_6	0	0	0	0	0	0	**1**

Figure 3-45. Reduced fault table

Example 3-16

In the fault table shown in *Figure 3-45* produced by fault simulation with fault dropping, only 19 faults need to be simulated compared to the case of 42 faults when simulation without fault dropping is carried out (the simulated faults in the fault table are shown in shadowed boxes). As the result of the fault dropping, however, the following faults remain not distinguishable: $\{F_2, F_3\}, \{F_1, F_4\}$.

3.4.2 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 observing also internal control points of the UUT (called also **guided probing**). Sequential diagnosis procedure can be graphically represented as **diagnostic tree**.

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 (*Figure 3-46*) of this process consists of the fault nodes FN (rectangles) and test nodes TN (circles). A FN is labelled by a set of not yet distinguished faults. The starting fault node is labelled by the set of all faults. To each FN_k a TN is linked which is labelled by a test pattern T_k to be applied as the next one. Every test pattern distinguishes between the faults that it detects and the ones it does not. The task of the test pattern T_k is to divide the faults in FN_k into two groups - detected and not detected by T_k 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 set of faults shown in a current fault node (rectangle) are equivalent (not distinguished) under the currently applied test set.

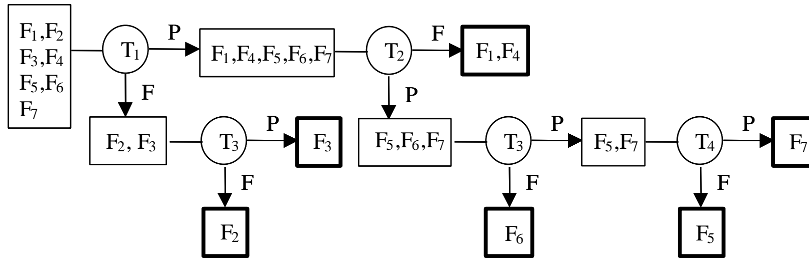


Figure 3-46. Diagnostic tree

<u>*Example 3-17*</u>

The diagnostic tree in *Figure 3-46* corresponds to the fault table in *Figure 3-43*. We can see that most of the faults are uniquely identified, two faults F_1 and F_4 remain indistinguishable. Not all test patterns used in the fault table are needed. Different faults are located by **identifying test sequences** with different lengths. The shortest test contains two patterns the longest one four patterns.

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 T_1 fails, the possible faults are $\{F_2,, F_3\}$. At this point applying T_2 would be wasteful, because T_2 does not distinguish between these faults. The use of adaptive testing may substantially decrease the average number of tests required to locate a fault.

Generating tests to distinguish faults. To improve the fault resolution of a given test set *T*, it is necessary to generate additional test patterns to distinguish among faults equivalent under the given test *T*.

Consider the problem of generating a test to **distinguish between faults** *F1* and *F2* [2]. Such a test must detect one of these faults but not the other, or vice versa. The following cases are possible:

- *F1* and *F2* do not influence the same set of outputs. Let *OUT(Fk)* be the set of outputs influenced by the fault *Fk*. A test should be generated for *F1* using only the circuit feeding the outputs *OUT(F1)*, or for *F2* using only the circuit feeding the outputs *OUT(F2)*.
- *F1* and *F2* influence the same set of outputs. A test should be generated for *F1* without activating *F2*, or vice versa, for F2 without activating *F1*.

Three possibilities can be mentioned to keep a fault F2: $x \equiv e$ not activated, where x denotes a line in the circuit, and $e \in \{0,1\}$:

- the value e should be assigned to the line x;
- if this is not possible then the activated path from *F2* should be blocked, so that the fault *F2* could not propagate and influence the activated path from *F1*;
- if the 2nd case is also not possible then the values propagated from the sites *F1* and *F2* and reaching the same gate *G* should be opposite to the inputs of *G*.

<u>*Example 3-18*</u>

Consider the following fault diagnosis cases in a gate-level circuit in *Figure 3-47*.

1. There are two faults in the circuit: *F1*: $x_{3,1}\equiv 0$, and *F2*: $x_4\equiv 1$. The fault *F1* may influence both outputs; the fault *F2* may influence only the output x_8. A test pattern 0010 activates *F1* up to the both outputs and *F2* to x_8 only. If both outputs will be wrong, *F1* is present, and if only the output x_8 will be wrong, *F2* is present.

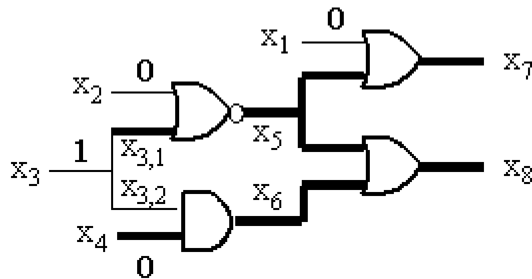


Figure 3-47. Gate-level circuit

2. There are two faults in the circuit: *F1*: $x_{3,2}\equiv 0$, and *F2*: $x_{5,2}\equiv 1$. Both of them influence the same output of the circuit. A test pattern 0100 activates the fault *F2*. The fault *F1* is not activated, because the line $x_{3,2}$ has the same value as it would have if *F1* were present.
3. There are the same two faults in the circuit: *F1*: $x_{3,2}\equiv 0$, and *F2*: $x_{5,2}\equiv 1$. Both of them influence the same output of the circuit. A test pattern 0110 activates the fault *F2*. The fault *F1* is activated at its site but not propagated through the AND gate, because of the value $x_4 = 0$ at its input.
4. There are two faults in the circuit: *F1*: $x_{3,1}\equiv 1$, and *F2*: $x_{3,2}\equiv 1$. A test pattern 1001 consists the value $x_1\equiv 1$ which creates the condition where both of the faults may influence only the same output x_8. On the other hand, the test pattern 1001 activates both of the faults to the same OR gate (i.e. none of them is blocked). However, the faults produce different values at the inputs of the gate, hence they are

distinguished. If the output value on x_8 will be 0, F1 is present. Otherwise, if the output value on x_8 will be 1, either F2 is present or none of the faults F1 and F2 are present.

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 physical location in the UUT. Probing is carried out step-by-step. In each step an internal signal is probed and compared to the expected value. The next probing depends on the result of the previous step.

A diagnostic tree can be created for the given test pattern to control the process of probing. The tree consists of internal nodes (circles) to mark the internal lines to be probed, and of terminal nodes (rectangles) to show the possible result of diagnosis. The results of probing are indicated as passed (P) or failed (F).

Typical faults located are "opens" and defective components. An open fault between two points A and B in a connection line is identified by a mismatch between the error observed at B and the correct value measured at A. A faulty device is identified by detecting an error at one of its outputs, while only correct values are measured at its inputs.

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 probings are available.

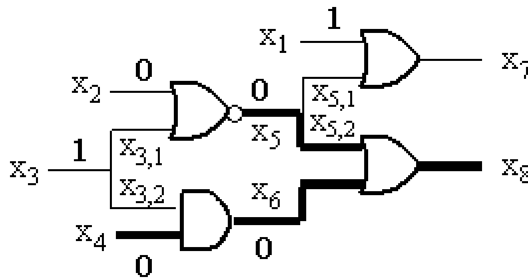


Figure 3-48. Gate-level circuit

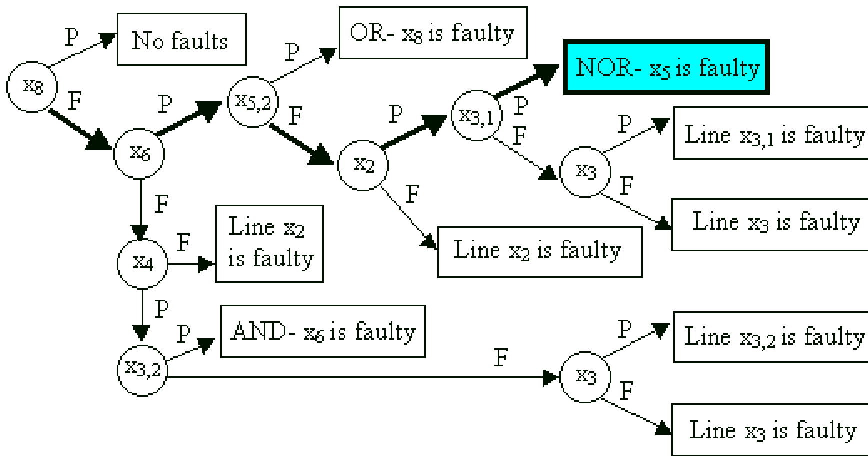


Figure 3-49. Diagnostic tree for guided probe testing

<u>*Example 3-19*</u>

Let us have a test pattern 1010 applied to the inputs of the circuit in *Figure 3-48*. The diagnostic tree created for this particular test pattern is shown in *Figure 3-49*. 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. Diagnostic tree allows carrying out optimization of the fault location procedure, for example, generating a procedure with minimum average number of probes.

3.5 TEST GENERATION FOR RAMS

The testing and operation of memories is radically different from logic. Memories are regular structures requiring special regular test patterns to perform testing chosen for a specific memory type. The complexity of memory testing is the numerous ways that a memory can fail. Many test patterns based on memory fault models (described in Chapter 2) are needed to test not only the cells but the peripheral circuitry around the memory cells as well. With a thorough examination of the specific transistor configurations utilised in the memory of concern, the appropriate fault models can be selected and the proper test patterns generated. No single pattern is sufficient to test a memory for all defect types. A set of patterns is needed to look for the real manufacturing defects and for interactions between the tightly packed adjacent memory structures. Memory testing is defect-based and algorithmic procedure.

Test sets in memory testing are now based on fault models and proven to have complete coverage for particular fault models and to be of minimal length for the given set of fault models covered by the tests. Numerous test algorithms for RAMs have been proposed over the years. Different types of memories require using different test algorithms [1], [14], [59], [60], [61].

3.5.1 Traditional memory testing

A wide variety of memory test sets based on different fault models has been developed. A memory test can be proposed based on requirement that every cell must be capable to storing both logical values 0 and 1, and to return the data when it is read. A memory test is a specific sequence of write and read operations applied to each cell of the memory cell array. For example, a simple test for single SA0 faults requires a sequence of write 1 (W1) and read logical 1 (R1) operations for every cell. For this reason we use the term **memory test algorithm** rather than memory test. One of the most important parameters of any memory testing is the number of test cycles needed to apply it, this is easily assessed by counting the number of read and write operations. So, the test algorithms are characterised by the test length, which determines the **test complexity** that vary from O(n) to $O(n^2)$, where n is the number of cells in the RAM chip.

The **traditional test algorithms** (such as *Zero-One, Galpat, Walking 1/0, Checkerboard, Sliding diagonal, Butterfly*), which are still being used, are less effective [60], [61]. They are still useful for detecting non-functional faults, such as refresh or sense amplifier recovery faults. Some of them provide more precise fault localisation than other algorithms.

Exhaustive. It is a test, in which all possible data combinations are included. If there are n memory cells then 2^n data combinations are possible; the original cell's state must be read, then re-written to the opposite state, and once more read to verify that state. It is clearly not feasible to perform an initial test which will confirm that any memory pattern can be stored and read correctly, since there are $3n2^n$ memory patterns theoretically possible.

Zero-One. First, zeros are written to all addresses and read from all addresses, and then ones are written to and read from all addresses. Each memory location is accessed four times, so the Zero-One test complexity is 4N - it is the number of all performed operations within the memory; where N is the number of address locations.

Zero-One test set achieves 100% stuck-at fault coverage of the memory cells but does not provide coverage for data retention, deceptive destructive read, address decoder faults (Zero-One test does not indicate whether each cell can be addressed uniquely).

Walkpat (*walking pattern*). A single cell (bit oriented memory) or a single address (word oriented memory) is in a different state (logical 0 or 1) from the other cells in the memory. The test set has a data background entirely of one data type (logical 0s or 1s), and a logical 1 (or 0) is walking through the memory so that to each address the test sequence ⇑ R0 (or 1), W1 (or 0), R1 (or 0), W0 (or 1) of read and write operations is applied. All four operations are performed on each address before proceeding to the next address. The addresses are selected incrementally from the zero address to the maximum address in the memory space as it is indicated by ⇑. The walking pattern has an execution time proportional to $2n^2$ where *n* is the number of cells (it is extremely long for large memory arrays). It checks memory for cell opens and shorts and address uniqueness.

Galpat (*galloping pattern*). The test proceeds as Walking test, except that after the 1 is stored in the first cell and while the other 0's are being checked, the first cell 1 is rechecked after each 0 is read, to ensure that the 1 remains undisturbed. As before, this sequence is repeated for every cell in the array and done with the complementary data.

Each succeeding cell then becomes the test cell in turn and the entire read process is repeated. All data is complemented and the entire test is repeated. Galpat has an execution time proportional $4n^2$, where *n* is the number of cells. It is effective for finding cell opens, shorts, address uniqueness faults, and sense amplifier interaction and access time problems.

All members of this class of test algorithms, characterised by a test time proportional to n^2, are unusable for large memory chips. An alternative, because the fault coverage is high, is to restrict "galloping" within a memory row or a column.

March test (*marching pattern*). It changes the data (a 0 or a 1) at a given address and leaves the address in the changed state when proceeding to the next address by applying the test sequence: ⇓ R0, W1, R1. (⇓ indicates that the address space successively decrements after performing each test sequence). This test assumes a zero background already existing in the memory.

3.5.2 Testing with March tests

Nowadays only those algorithms, which test complexity increases linearly with the number of memory cells, are of importance for memory testing. The use of newer algorithms gives shorter execution time and also better fault coverage.

This subsection is concerned with layout-independent RAM testing only. We abstract a defect model for the RAM, which is based on the most likely layout and design defects, into a functional fault model, and this leads to the

set of reduced functional faults [1], [60]. The set includes *stuck-at faults* (SAF), *transition faults* (TF), *address decoder faults* (AF), *coupling faults* (CF) and *neighbourhood pattern sensitive faults* (NPSF). The fault models are described in Chapter 2.

3.5.3 Testing stuck-at, transition, address and coupling faults

The simplest tests, which detect SAFs, TFs and CFs are part of a family of March type test algorithms that are in present the most preferred algorithms for RAM array testing. The March tests are of the Nth order, which make them fast.

A **March test algorithm** consists of a sequence of March test elements. A **March test element** is a finite sequence of write and/or read operations (W0/W1, R0/R1) applied consecutively to a cell in the memory array. Then applied to the next cell until all cells have been treated. The addresses of the next cells are determined either in increasing (⇑) or decreasing address order (⇓) or the address order is irrelevant (⇕). After applying one March element to each cell, the next March element of the March test algorithm is taken. There is one requirement that the increasing and decreasing address orders during performing one March test algorithm have to be always inverse. The length of a March test algorithm is defined as the number of March elements multiplied by the number of memory cells.

An example is shown in *Figure 3-50*: the MATS algorithm (Modified algorithmic test sequence) has three March test elements: ⇕W0; ⇑R0,W1; ⇕R1,W0). *Figure 3-51* shows how the SA0 fault in the cell with address (2,1) is detected by MATS+ March test. The fault is detected by the element 2 (⇑R0,W1) as it moves from the highest memory address downward and expects to read a 1 in cell (2,1), but gets a 0 instead. *Figure 3-52* shows how MATS+ detects the multiple address decoder faults, where cell (2,1) is unaddressable, and address (2,1) maps to an access of cell (3,1). Since all writes to cell (2,1) have no effect, and any read of cell (2,1) produces a random result, the defective cell will be detected either by March element 1 when it reads cell (2,1) if the read returns a 1 when a 0 was expected, or by element 2 when it reads cell (2,1) if the read returns a 0 when a 1 was expected. March element 1 writes a 1 to cell (2,1) but that has the effect of writing cell (3,1). This is detected when element 1 operates on cell (3,1), because it first expects to read a 0 but gets an unexpected 1, and then it writes a 1 to the cell. If the address of cell (3,1) mapped into an access of cell (2,1), then March element 2 would detect this fault as it descended from highest to lowest addresses in memory. It would expect to read a 1 from cell (2,1), but would get a 0 instead.

Element	Operation
1	⇕W0
2	⇑R0, W1
3	⇓R1, W0

Figure 3-50. MATS+ test algorithm

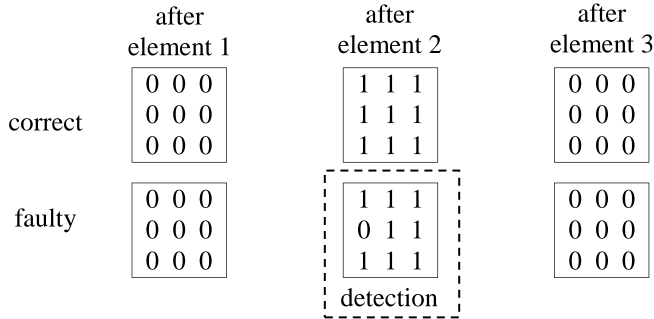


Figure 3-51. Detection of SAF0 by MATS+ test: {⇕W0; ⇑R0,W1; ⇓R1,W0} [1]

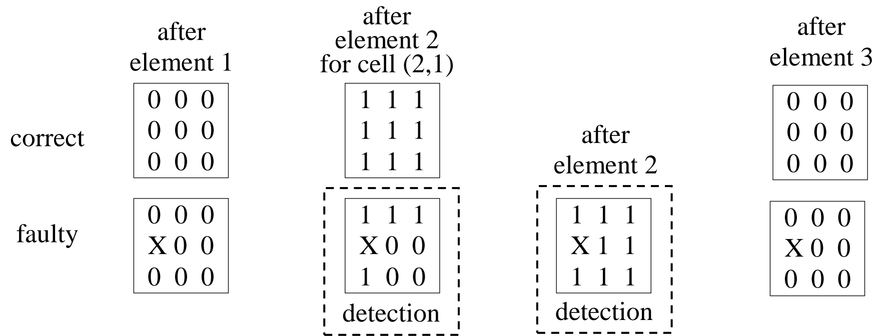


Figure 3-52. Detection of multiple AFs by MATS+ test [1]

It was proven [1], [60] that a March test detects all AFs if it satisfies two conditions:

- the value x must be read successively from all cells and the value non-x must be written successively to all cells following the increasing address order;
- the value non-x must be read successively from all cells and the value x must be written successively to all cells following the decreasing address order.

Table 3-13. March type test algorithms (FC: fault coverage, (x N): complexity) [60], [61]

	MATS (4 N)	**MATS+** (5 N)	**MATS++** (6 N)
1	⇕ W0	⇕ W0	⇕ W0
2	⇕ R0, W1	⇑ R0, W1	⇑ R0, W1
3	⇕ R1	⇓ R1, W0	⇓ R1, W0, R0
FC:	SAF	SAF, AF	SAF, AF, TF
	March X (6 N)	**March Y** (8N)	
1	⇕ W0	⇕ W0	
2	⇑ R0, W1	⇑ R0, W1, R1	
3	⇓ R1, W0	⇓ R1, W0, R0	
4	⇕ R0	⇕ R0	
FC:	SAF, AF, TF, CFin	SAF, AF, TF, CFin, TF linked with CFin	
	March A (15 N)	**March B** (17 N)	**PMOVI** * (13 N)
1	⇕ W0	⇕ W0	⇓ W0
2	⇑ R0, W1, W0, W1	⇑ R0, W1, R1, W0, R0, W1	⇑ R0, W1, R1
3	⇑ R1, W0, W1	⇑ R1, W0, W1	⇑ R1, W0, R0
4	⇓ R1, W0, W1, W0	⇓ R1, W0, W1, W0	⇓ R0, W1, R1
5	⇓ R0, W1, W0	⇓ R0, W1, W0	⇓ R1, W0, R0
FC:	SAF, AF, TF, CFin linked CFid	SAF, AF, TF, linked TF, CFin, linked CFid	
	Marching 1/0 (14 N)	**March C-** (10 N)	**Enhanced March C-** (18 N)
1	⇑ W0	⇕ W0	⇕ W0
2	⇑ R0, W1, R1	⇑ R0, W1	⇑ R0, W1, R1, W1
3	⇓ R1, W0, R0	⇑ R1, W0	⇑ R1, W0, R0, W0
4	⇑ W1	⇓ R0, W1	⇓ R0, W1, R1, W1
5	⇑ R1, W0, R0	⇓ R1, W0	⇓ R1, W0, R0, W0
6	⇓ R0, W1, R1	⇕ R0	⇕ R0
FC:	SAF, AF	SAF, AF, TF, CFs	SAF, AF, TF, CFs, pre-charge defects
	March LR (14 N)	**March LA** (22 N)	**March SR+** (18 N)
1	⇕ W0	⇕ W0	⇓ W0
2	⇓ R0, W1	⇑ R0, W1, W0, W1, R1	⇑ R0, R0, W1, R1, R1,W0,R0
3	⇑ R1, W0, R0, W1	⇑ R1, W0, W1, W0, R0	⇓ R0
4	⇑ R1, W0	⇓ R0, W1, W0, W1, R1	⇑ W1
5	⇑ R0, W1, R1, W0	⇓ R1, W0, W1, W0, R0	⇓ R1, R1, W0, R0, R0,W1,R1
6	⇑ R0	⇓ R0	⇑ R1
FC:	linked faults	all simple, many linked faults	SAF, TF, CFs, dec.destr.reads
	March C (11 N)	**March SRD+** (18 N)	**March G** (23 N)
1	⇕ W0	⇓ W0	⇕ W0
2	⇑ R0, W1	⇑ R0, R0, W1, R1, R1,W0,R0	⇑ R0, W1, R1, W0, R0, W1
3	⇑ R1, W0	Pause	⇑ R1, W0, R1
4	⇕ R0	⇓ R0	⇓ R1, W0, W1, W0
5	⇓ R0, W1	⇑ W1	⇓ R0, W1, W0
6	⇓ R1, W0	⇓ R1, R1, W0, R0, R0,W1,R1	Pause
7	⇕ R0	Pause	⇕ R0, W1, R1
8		⇑ R1	Pause
9		FC: SAF, TF, CFs, retention	⇕ R1, W0, R0
FC:		faults, decept. destruct. reads	SOF, retention faults

**Partial moving inversion*

– Specific test sets with their sequence of operations and covered fault models, which are widely used in the industry to find manufacturing defects, are listed in easing address order.

Table 3-13.

Partial moving inversion test set checks, if the data has been correctly stored into the cell by the third read operation performed on each cell immediately after the write operation to prevent defect masking.

Detecting of pre-charge defects is accomplished by the rapid succession of the fourth operation in an element of Enhanced March C- pattern. A defect can prevent the bit lines from pre-charging correctly. The same column must be utilised while different rows are addressing successively.

March LR pattern is a combination of marching and walking elements and was developed to detect realistic linked faults. March G pattern includes a pause in the sequence to facilitate retention testing.

The March A+ and March A++ patterns and also March C+ and March C++ patterns are extensions of the March A and March C pattern respectively. In the March A+ and March C+ each read operation is replaced by three read operations, which allows to detect pull-up and pull-down paths, which are disconnected in a cell. The March A++ and March C++ patterns include two delay elements, which allows detecting retention defects.

3.5.4 Testing word-oriented memories

Memories that have single bit data input only and data output are **bit-oriented**, and each memory cell can be addressed individually. Memories, which have wider data input and data output buses, are **word-oriented** memories, and when a read or write operation to an address is performed, the full width of the data bus is utilised.

Patterns developed in the past for a bit-oriented memory have to be modified for the word orientation. In a word-oriented memory a single cell cannot be addressed individually. Possible interactions between cells-bits within a word are covered by the tests, only if various data background patterns are used.

The number of data background patterns required is $\log_2 m+1$, where m is the number of bits in a word. Thus a memory with an 8-bit word requires four data background patterns, and it is always assumed that a background and its complementary background are to be utilised (an example is seen in *Table 3-14*).

Other data background test requirements can be based also on the cell adjacencies.

Table 3-14. Data backgrounds and their inverses for an 8-bit word [61]

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 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
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

3.5.5 Testing neighbourhood pattern sensitive faults

Tests for NPSFs (the NPSF model is explained in Chapter 2) cannot be performed by March tests because the base cell has to be treated differently from other cells of the neighbourhood. On the other hand, the NPSF tests do not detect AFs. It is always assumed that read operations of memory cells are fault-free in the NPSF testing (can be ensured by a March test).

To test a neighbourhood for a certain kind of NPSF, all required test patterns (*Table 3-15*) must be applied to that neighbourhood, and after each test pattern the base cell must be read. In this way all NPSFs can be not only detected but also located.

Active neighbourhood patterns (ANPs). The required test patterns for testing ANPSFs are composed of:

– base cell is in two different possible values (0 and 1);
– one of the deleted neighbourhood cells undergoes up and down transitions;
– other deleted neighbourhood cells are in all combinations of the logical values 0 and 1.

The total number of ANPs is $(k-1) \cdot 2^k$ (where k is the size of the neighbourhood).

Passive neighbourhood patterns (PNPs). Test patterns for PNPSFs are:

– base cell undergoes up and down transitions;
– other deleted neighbourhood cells are in all combinations of the logical values 0 and 1.

There are 2^k PNPs.

Static neighbourhood patterns (SNPs). The number of test patterns for SNPSFs is determined by all possible combinations of the given neighbourhood cells logic values and is also equal 2^k.

It is important to minimise the number of write operations during NPSF testing, in order to obtain the shortest possible test. The mechanisms how to apply a sequence of all required patterns and performed a minimal number of write operations is based on the fact that the difference between a pattern and its successor has to be minimal, it means that they should differ in one bit.

Table 3-15. All possible NPSF test patterns for the type 1 neighbourhood: b=base cell, d_1,d_2,d_3,d_4=deleted neighbourhood cells [1], [60]

ANPs			
b	00000000000000001111111111111111	b	00000000000000001111111111111111
d_1	↑↑↑↑↑↑↑↑↓↓↓↓↓↓↓↓↑↑↑↑↑↑↑↑↓↓↓↓↓↓↓↓	d_1	00001111000011110000111100001111
d_2	00001111000011110000111100001111	d_2	00110011001100110011001100110011
d_3	00110011001100110011001100110011	d_3	↑↑↑↑↑↑↑↑↓↓↓↓↓↓↓↓↑↑↑↑↑↑↑↑↓↓↓↓↓↓↓↓
d_4	01010101010101010101010101010101	d_4	01010101010101010101010101010101
b	00000000000000001111111111111111	b	00000000000000001111111111111111
d_1	00001111000011110000111100001111	d_1	00001111000011110000111100001111
d_2	↑↑↑↑↑↑↑↑↓↓↓↓↓↓↓↓↑↑↑↑↑↑↑↑↓↓↓↓↓↓↓↓	d_2	00110011001100110011001100110011
d_3	00110011001100110011001100110011	d_3	01010101010101010101010101010101
d_4	01010101010101010101010101010101	d_4	↑↑↑↑↑↑↑↑↓↓↓↓↓↓↓↓↑↑↑↑↑↑↑↑↓↓↓↓↓↓↓↓
PNPs		**SNPs**	
b	↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓	b	00000000000000001111111111111111
d_1	00000000111111110000000011111111	d_1	00000000111111110000000011111111
d_2	00001111000011110000111100001111	d_2	00001111000011110000111100001111
d_3	00110011001100110011001100110011	d_3	00110011001100110011001100110011
d_4	01010101010101010101010101010101	d_4	01010101010101010101010101010101

A *k*-bit **Eulerian graph** (in *Figure 3-53* a 3-bit Eulerian graph is depicted) is defined as a graph in which there is a node for each *k*-bit pattern of 1s and 0s and there is an edge between two nodes, if they differ just in one bit. The edges in the Eulerian graph correspond to the ANPs and PNPs of a *k*-bit neighbourhood, and the nodes correspond to SNPs. An Eulerian sequence is a sequence through the Eulerian graph which traverses each edge just once and which should be used in the case that there are up or down transitions in the patterns. A **Hamiltonian sequence** on the other hand traverses each node of the Eulerian graph just once and should be used if the patterns contain only 0s and 1s.

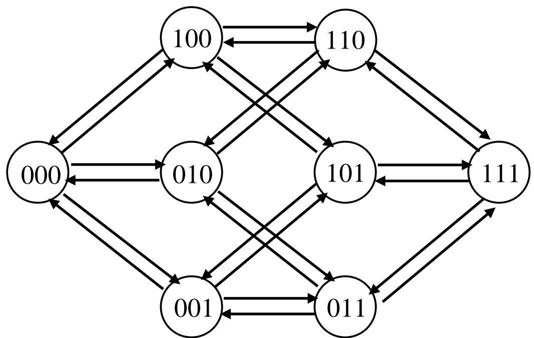


Figure 3-53. Eulerian graph for 3-bit patterns [1]

When data are written into the base cell is, we change k different neighbourhoods (type 1 or type 2) and we wish to test the neighbourhoods simultaneously, using two methods for this purpose: *tiling* and *two-group*.

The **tiling method** totally covers memory with non-overlapping neighbourhoods as it is shown in *Figure 3-54*. Memory cell 2 is always the base cell and other numbered cells are deleted neighbourhood cells. Now there are n/5 base cells to which all the test patterns are applied. It turns out that also appropriate patterns are applied to the memory when cell 0, cell 1, cell 3 or cell 4 is the base cell. This reduces the pattern length from n . 2^k patterns to n/k . 2^k patterns while each cell is simultaneously a base cell and a deleted neighbourhood cell for other base cells.

0	1	b	3	4	0	1	b	3	4
b	3	4	0	1	b	3	4	0	1
4	0	1	b	3	4	0	1	b	3
1	b	3	4	0	1	b	3	4	0
3	4	0	1	b	3	4	0	1	b
0	1	b	3	4	0	1	b	3	4
b	3	4	0	1	b	3	4	0	1
4	0	1	b	3	4	0	1	b	3
1	b	3	4	0	1	b	3	4	0
3	4	0	1	b	3	4	0	1	b

Figure 3-54. Five cells (type1) tiling neighbourhood [1], [60]

In the case of the **two-group method**, a cell is simultaneously a base cell in one group and a deleted neighbourhood cell in the second group (*Figure 3-55*). The memory cells are divided into two groups in a **checkerboard pattern**, i.e. the base cells in group 1 become deleted neighbourhood cells in group 2, and vice versa. Each group has n/2 base cells (denoted as b) and n/2 deleted neighbourhood cells of divided into four subgroups d_1, d_2, d_3 and d_4.

d_1	b	d_2	b	d_1	b	d_2	b
b	d_3	b	d_4	b	d_3	b	d_4
d_2	b	d_1	b	d_2	b	d_1	b
b	d_4	b	d_3	b	d_4	b	d_3
d_1	b	d_2	b	d_1	b	d_2	b
b	d_3	b	d_4	b	d_3	b	d_4
d_2	b	d_1	b	d_2	b	d_1	b
b	d_4	b	d_3	b	d_4	b	d_3

b	d_1	b	d_2	b	d_1	b	d_2
d_3	b	d_4	b	d_3	b	d_4	b
b	d_2	b	d_1	b	d_2	b	d_1
d_4	b	d_3	b	d_4	b	d_3	b
b	d_1	b	d_2	b	d_1	b	d_2
d_3	b	d_4	b	d_3	b	d_4	b
b	d_2	b	d_1	b	d_2	b	d_1
d_4	b	d_3	b	d_4	b	d_3	b

Figure 3-55. Labels of cells in the two-group method [1]

Table 3-16 gives NPSF testing algorithms overview and some performance details as the type of the used neighbourhood, the number of cells involved in the neighbourhood, method used for the simultaneous testing, which fault models are covered (detected or also located, the operation count).

No single type of test (March, NPSF, DC parametric, AC parametric) is sufficient for current RAM testing needs, so a combination of various test is used.

Table 3-16. Overview of NPSF testing algorithms (L=location, D=detection) [1], [60]

Algorithm	Neighbourhood/*k*/Method	Fault coverage					Complexity
		SAF	TF	NPSF			
				A	P	S	
TD ANPSF 1G	type 1 / 5 / 2 group	L		D			163.5 n
TL ANPPSF 1G	type 1 / 5 / 2 group	L	L	L	L	L	195.5 n
TL ANPPSF 2T	type 2 / 9 / tiling	L	L	L	L		5122 n
TL ANPPSF 1T	type 1 / 5 / tiling	L	L	L	L		194 n
TL SNPSF 1G	type 1 / 5 / 2 group	L				L	43.5 n
TL SNPSF 1T	type 1 / 5 / tiling	L				L	39.2 n
TL SNPSF 2T	type 2 / 9 / tiling	L				L	569 n
TD SNPSF 1G	type 1 / 5 / 2 group	L				D	36.125 n

3.5.6 Testing RAM technology and layout related faults

The coupling fault tests may not be effective because the DRAMs may be repaired after manufacturing testing or DRAM address lines are scrambled. Also, the G-bit DRAMs have new kind of defects.

With deep sub-micron chip feature sizes, memory chips are increasingly subject to peculiar, layout specific failures. Therefore, inductive fault analysis (IFA) is now used to analyse the chip layout and determine which fault models correctly model the actual physical defects that may occur. IFA is now necessary to ensure that the actual defects that occur are mapped into a fault model, and appropriate tests can be selected for that fault model. After performing IFA faults caused by actual defects as broken wires, shorts between wires, missing contacts, extra contacts, parasitic transistors can be found. These defects can be mapped to functional faults as SAF, SOF, TF in a memory cell or state CF and CFid between two cells and also data retention fault caused by broken pull-up device. The March type tests IFA-9 and IFA13 were extended by a new March element Delay, which means to wait for 100 ms to be able to test for data retention faults (see *Table 3-17*).

Table 3-17. Overview of the IFA testing algorithms [60], [61]

	IFA-9 (12 N + Delays)	**IFA-13** (16N+ Delays)

1	⇕W0	⇕W0
2	⇑R0, W1	⇑R0, W1, R1
3	⇑R1, W0	⇑R1, W0, R0
4	⇓R0, W1	⇓R0, W1, R1
5	⇓R1, W0	⇓R1, W0, R0
6	Delay	Delay
7	⇑R0, W1	⇑R0, W1
8	Delay	Delay
9	⇑R1	⇑R1
FC:	SAF, TF, AF, CFid, data retention fault	SAF, TF, AF, CFs for bits in the same word, SOF, data retention

REFERENCES

1. Bushnell M. L., Agrawal V. D. Essential of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits, Kluwer Academic Publisher, 2000.
2. Abramovici M., Breuer M.A., Friedman A.D. Digital Systems testing and testable Design, Computer Science Press, 1995.
3. Hurst S.L. VLSI Testing Digital and Mixed Analogue-Digital techniques, The Institution of Electrical Engineers, 1998.
4. Sachdev M. Defect Oriented Testing for CMOS Analog and Digital Circuits. Kluwer Academic Publisher, 1998.
5. Cibáková T., Fischerová M., Gramatová E., Kuzmicz W., Pleskacz W., Raik J., Ubar R. Hierarchical Test Generation for Combinational Circuits with Real Defects Coverage. In: Journal of Microelectronics Reliability 42 Pergamon Press, 2002, pp. 1141-1149.
6. Maxwell P., Aitken R. Defect-Oriented Testing. The IEEE ETW'03 (European Test Workshop) tutorial, Maastricht, 2003.
7. Goel P.An Implicit Enumeration Algorithm to Generate tests for Combinational Logic Circuits. In IEEE Transactions on Computers, vo. C-30, no. 3, pp.215-222, 1981.
8. Roth J.P.: Diagnosis of Automata Failures: Calculus and a Method. In: IBM Journal of research and Development, vol. 10, no. 4, pp. 278-291, 1966.
9. Fujiwara H.: Logic testing and Design for testability. Cambridge, Massachusetts, MIT Press, 1985.
10. Schulz M.H., Trischler E., Bryant R.E.: SOCRATES: A Highly Efficient Automatic Test Pattern generation System. In IEEE Transactions on Computer -Aided Design, vo. CAD-7, 1988, pp. 126-137.
11. Kjelkerud E., Mercer M.R: A Topological Search Algorithm for ATPG. In: Proc. of the 24th Design Automation Conference, 1987, pp. 502-508.
12. Giraldi J., Bushnell M.L.: EST: The new Frontier in Automatic Test Pattern Generation. In: Proc. of the 27th International test Conference, 1991, pp. 662-672.
13. Kunz W., Pradham D.K.: Recursice Learning: An Attractive Alternative to the Decision Tree for test Generation in Digital Circuits. In: Proc. of the International test Conference, 1992, pp. 826-825.
14. Crouch A. L. *Design for Test for Digital IC's and Embedded Core Systems*, Prentice Hall PTR, New Jersey, USA, 1999, 347 p.
15. Lala P.K.: Digital Circuits Testing and testability, Academic press, 1997.
16. Gizdarski E., Fujiwara H.: SPIRIT: A Highly Robust Combinational Test Generation Algorithm. In: Proc. of International Test Conference.
17. Hamzaoglu I., Patel J.H. Patel: New Techniques for Deterministic Test Pattern Generation, Proc. of VTS'98, pp.lee S., Cobb B., Dworak J., Grimaila M.R. and Mercier

M.R.: A New ATPG Algorithm to Limit test Size and Achieve Multiple Detections of all Faults, Proc. of DATE'02, pp. 94-99.
18. Tsai K.-H., Tompson, Rajski J., Sadowaska M.M: STAR-ATPG: A High Speed Test Pattern Generator for Large Scan Design., Proc. of ITC'99, pp. 1021-1030.
19. Wang, Ch., Reddy S.M., Pomeranz I., Lin X., Rajski J.: Conflict Driven Techniqueas for Improving Deterministic Test Pattern Generation, Proc. of ICCAD'2002, pp.
20. Wang, Z., Sadowaska M.M., Rajski J.: Defect Behavior Extraction Using Stuck_at Fault Model, Proc. of DDECS'03, pp. 239-244.
21. Chakravarty S., Thadkaran P. J.: Introduction to I_{DDQ} Testing. Kluwer Academic Publisher, 1997.
22. Chakravarty S.: Defect Based Testing. IEEE DDECS'01 (Design and Diagnostics of Electronic Circuits and Systems) Workshop, Györ, 2001, invited talk.
23. Lee C.Y. Representation of Switching Circuits by Binary Decision Programs. The Bell System Technical Journal, July 1959, pp.985-999.
24. Ubar R. Test Generation for Digital Circuits with Alternative Graphs. Proceedings of Tallinn Technical University No 409, 1976, pp.75-81 (in Russian).
25. Akers S.B. Functional Testing with Binary Decision Diagrams. J. of Design Automation and Fault-Tolerant Computing, Vol.2, Oct. 1978, pp.311-331.
26. Plakk M., Ubar R. Digital Circuit Test Design using the Alternative Graph Model. Automation and Remote Control, Vol.41, No 5, part 2, Nov. 1980, Plenum Publishing Corporation, USA, pp. 714-722.
27. Ubar R. Vektorielle Alternative Graphen und Fehlerdiagnose für digitale Systeme. Nachrichtentechnik/Elektronik, (31) 1981, H.1, pp.25-29.
28. Ubar R. Test Generation for Digital Systems on the Vector Alternative Graph Model. Proc. of the 13th Annual Int. Symp. on Fault Tolerant Computing, Milano, Italy, 1983, pp.374-377.
29. Bryant R.E. Graph-based algorithms for Boolean function manipulation. IEEE Trans. on Comp., 35 (8): 677-691, 1986.
30. Minato S. Binary Decision Diagrams and Applications for VLSI CAD. *Kluwer Academic Publishers,* 1996, 141 p.
31. Ubar R. Test Synthesis with Alternative Graphs. *IEEE Design and Test of Computers.* Spring, 1996, pp.48-59.
32. Drechsler R., Becker B. Binary Decision Diagrams. Kluwer Academic Publishers, 1998, 200 p.
33. Ubar R. Multi-Valued Simulation of Digital Circuits with Structurally Synthesized Binary Decision Diagrams. *OPA (Overseas Publishers Assotiation) N.V. Gordon and Breach Publishers, Multiple Valued Logic,* Vol.4 pp. 141-157, 1998.
34. Raik J., Ubar R.. Feasibility of Structurally Synthesized BDD Models for Test Generation. *Proc. of the IEEE European Test Workshop,* Barcelona (Spain), May 27-29, 1998, pp.145-146.
35. Mourad S., Zorian Y. Principles of Testing Electronic Systems. J.Wiley & Sons, Inc. New York, 2000, 420 p.
36. Gupta A., Armstrong J.R. Functional fault modeling and simulation for VLSI devices. ACM/IEEE 22nd Design Automation Conference, 1985, pp.720-726.
37. Santucci J.F., Courbis A.L., Giambiasi N. Speed up of behavioral ATPG. using a heuristic criterion, 30th ACM/IEEE Design Automation Conference, pp. 92-96, 1993.
38. Cho C.H., Armstrong J.R. B-algorithm: A Behavioral Test Generation Algorithm, IEEE 1994 International Test Conference, pp.968-979.
39. Bhattacharya D., Hayes J.P. A hierarchical test generation methodology for digital circuits, JETTA: Theory and Application, vol. 1, pp. 103-123, 1990.
40. Karam M., Leveugle R., Saucier G. Hierarchical test generation based on delayed propagation, IEEE 1991 International Test Conference, pp.739-747.

41. Lee J., Patel J.H. Hierarchical test generation under intensive global functional constraints, 29thACM/IEEE Design Automation Conf., June 1992, pp. 261-266.
42. Yadavalli S., Pomeranz I., Reddy S.M. MUSTC-Testing: Multi-Stage-Combinational Test Scheduling at the Register-Transfer Level, 8th Int. Conference on VLSI Design, January 1995, pp. 110-115.
43. Abadir M.S. et al. A Knowledge-Based System, IEEE Design & Test, August 1985, pp.56-68.
44. Annaratone M.A., Sami M.G. An Approach to Functional Testing of Microprocessors. Digest of Papers 12th Annual Int. Symp. On Fault-Tolerant Computing, June 1982, pp.158-164.
45. Abadir M.S.,.Reghbati H.K. Functional Specification and Testing of Logic Circuits. Comp. & Math. With Appls, Vol.11, No 12, 1985, pp.1143 – 1153.
46. Su S.Y.H., Lin T.. Functional Testing Techniques for Digital LSI/VLSI Systems. Dig. of papers 21st IEEE Design Automation Conference, 1984, pp.517-528.
47. Freeman S. Test Generation for Data Path. IEEE J. of Solid-State Circuits, Vol.23, April 1988, pp.421-427.
48. Abadir M.S., Reghbati H.K. Test Generation for LSI: A Case Study. Proc. of 21st Design Automation Conf., June 1984, pp.180-195.
49. Ubar R., Moraviec A., Raik J. Cycle-based Simulation with Decision Diagrams. IEEE Proc. of Design Automation and Test in Europe. Munich, March 9-12, 1999, pp.454-458.
50. Raik J., Ubar R. Sequential Circuit Test Generation Using Decision Diagram Models. IEEE Proc. of Design Automation and Test in Europe. Munich, March 9-12, 1999, pp. 736-740.
51. Raik J.,.Ubar R. Fast Test Pattern Generation for Sequential Circuits Using Decision Diagram Representations. Journal of Electronic Testing: Theory and Applications. Kluwer Academic Publishers. Vol. 16, No. 3, pp. 213-226, 2000.
52. Ubar R. Representing Transparency Conditions in Test Generation for VLSI by Decision Diagrams. The 1st Electronic Circuits and Systems Conference. Bratislava, September 4-5, 1997, pp.213-216.
53. Brahme D., Abraham J.A. Functional Testing of Microprocessors. IEEE Trans. on Computers, Vol. C-33, June 1984, pp.475-485.
54. Shen L., Su S.Y.H.. A Functional Testing Method for Microprocessors. IEEE Trans. on Computers, Vol. 37, No 10, Oct 1988, pp.1288-1293.
55. Thatte S.M., Abraham J.A. Test Generation for Microprocessors. IEEE Trans. on Computers, Vol. C-29, No 6, 1980, pp.429-441.
56. Abraham J.A., Parker K.P. Practical Microprocessor Testing: Open and Closed Loop Approaches. Proc. COMPCON Spring 1981, pp.308-311.
57. Ubar R., Raik J., Ivask E., Brik M. Multi-Level Fault Simulation of Digital Systems on Decision Diagrams. IEEE Workshop on Electronic Design, Test and Applications – DELTA'02, Christchurch, New Zealand, 29-31 January 2002, pp.86-91.
58. Thaise A. Boolean Differential Calculus. Philips Res. Repts., Vol. 26, 1971, pp. 229-246.
59. Mazumder P., Chakraborty K. Testing and Testable Design of High-Density Random Access Memories, Kluwer Academic Publishers, 1996.
60. Van de Goor A.J. Testing Semiconductor Memories. Theory and Practice, ComTex Publishing, Gouda, The Netherlands, 1998, 512 p.
61. Adams R. Dean. High Performance Memory Testing. Design Principles, Fault Modeling and Self-Test, Kluwer Academic Publishers, 2003, 246 p.