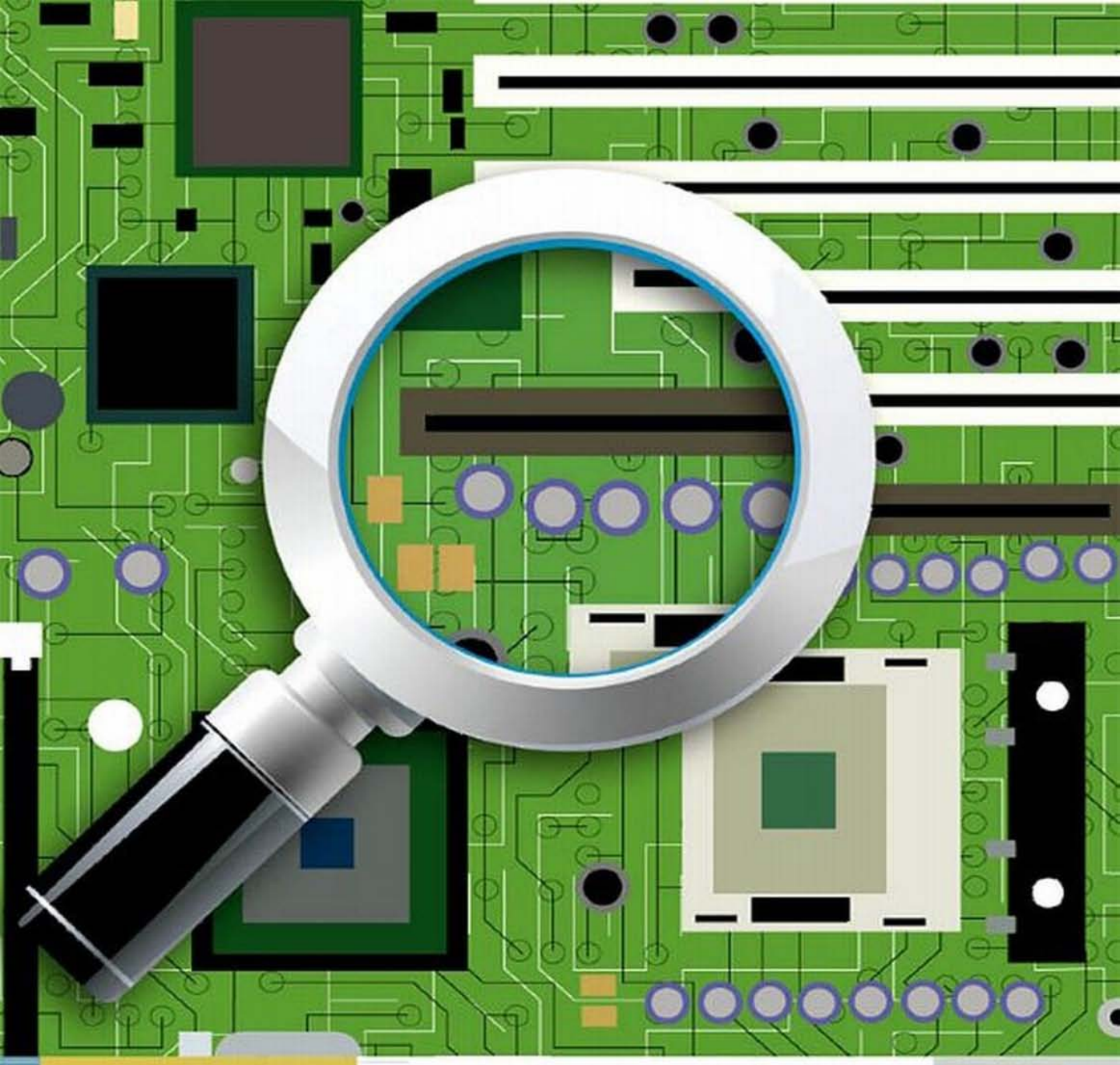


SYNTHESIS, VERIFICATION, AND TEST

# ELECTRONIC DESIGN AUTOMATION

SYSTEMS ON SILICON

LAUNG-TERNG WANG • YAO-WEN CHANG
KWANG-TING (TIM) CHENG

MORGAN KAUFMANN

# Contents

## 1.3 TEST AUTOMATION

Advances in manufacturing process technology have also led to very complex designs. As a result, it has become a requirement that *design-for-testability* (DFT) features be incorporated in the ***register-transfer level*** (RTL) or gate-level design before physical design to ensure the quality of the fabricated devices. In fact, the traditional VLSI development process illustrated in Figure 1.3 involves some form of testing at each stage, including design verification. Once verified, the VLSI design then goes to fabrication and, at the same time, test engineers develop a test procedure based on the design specification and **fault models** associated with the implementation technology. Because the resulting product quality is in general unsatisfactory, modern VLSI test development planning tends to start when the RTL design is near completion. This test development plan defines what **test requirements** the product must meet, often in terms of **defect level** and **manufacturing yield**, test cost, and whether it is necessary to perform self-test and diagnosis. Because the test

requirements mostly target manufacturing defects rather than **soft errors**, which would require **online fault detection and correction** [Wang 2007], one need is to decide what fault models should be considered.

The test development process now consists of (1) defining the targeted fault models for defect level and manufacturing yield considerations, (2) deciding what types of DFT features should be incorporated in the RTL design to meet the test requirements, (3) generating and fault-grading test patterns to calculate the final fault coverage, and (4) conducting manufacturing test to screen bad chips from shipping to customers and performing ***failure mode analysis*** (FMA) when the chips do not achieve desired defect level or yield requirements.

### 1.3.1 Fault models

A **defect** is a manufacturing flaw or physical imperfection that may lead to a **fault**, a fault can cause a circuit **error**, and a circuit error can result in a **failure** of the device or system. Because of the diversity of defects, it is difficult to generate tests for real defects. Fault models are necessary for generating and evaluating test patterns. Generally, a good fault model should satisfy two criteria: (1) it should accurately reflect the behavior of defects and (2) it should be computationally efficient in terms of time required for fault simulation and test generation. Many fault models have been proposed but, unfortunately, no single fault model accurately reflects the behavior of all possible defects that can occur. As a result, a combination of different fault models is often used in the generation and evaluation of test patterns. Some well-known and commonly used fault models for general sequential logic [Bushnell 2000; Wang 2006] include the following:

1. **Gate-level stuck-at fault model**: The stuck-at fault is a logical fault model that has been used successfully for decades. A stuck-at fault transforms the correct value on the faulty signal line to appear to be stuck-at a constant logic value, either logic 0 or 1, referred to as ***stuck-at-0*** (SA0) or ***stuck-at-1*** (SA1), respectively. This model is commonly referred to as the **line stuck-at fault model** where any line can be SA0 or SA1, and also referred to as the gate-level stuck-at fault model where any input or output of any gate can be SA0 or SA1.
2. **Transistor-level stuck fault model**: At the switch level, a transistor can be **stuck-off** or **stuck-on**, also referred to as **stuck-open** or **stuck-short**, respectively. The line stuck-at fault model cannot accurately reflect the behavior of stuck-off and stuck-on transistor faults in ***complementary metal oxide semiconductor*** (CMOS) logic circuits because of the multiple transistors used to construct CMOS logic gates. A stuck-open transistor fault in a CMOS combinational logic gate can cause the gate to behave like a level-sensitive latch. Thus, a stuck-open fault in a CMOS combinational circuit requires a sequence of two vectors for

detection instead of a single test vector for a stuck-at fault. Stuck-short faults, on the other hand, can produce a conducting path between power ($V_{DD}$) and ground ($V_{SS}$) and may be detected by monitoring the power supply current during steady state, referred to as $I_{DDQ}$. This technique of monitoring the steady state power supply current to detect transistor stuck-short faults is called **$I_{DDQ}$ testing** [Bushnell 2000; Wang 2007].

3. **Bridging fault models**: Defects can also include opens and shorts in the wires that interconnect the transistors that form the circuit. Opens tend to behave like line stuck-at faults. However, a **resistive open** does not behave the same as a transistor or line stuck-at fault, but instead affects the propagation delay of the signal path. A short between two wires is commonly referred to as a **bridging fault**. The case of a wire being shorted to $V_{DD}$ or $V_{SS}$ is equivalent to the line stuck-at fault model. However, when two signal wires are shorted together, bridging fault models are needed; the three most commonly used bridging fault models are illustrated in Figure 1.9. The first bridging fault model proposed was the **wired-AND/wired-OR** bridging fault model, which was originally developed for bipolar technology and does not accurately reflect the behavior of bridging faults typically found in CMOS devices. Therefore, the **dominant bridging fault** model was proposed for CMOS where one driver is assumed to dominate the logic value on the two shorted nets. However, the dominant bridging fault model does not accurately reflect the behavior of a resistive short in some cases. The most recent bridging fault model, called the **4-way** bridging fault model and also known as the **dominant-AND/dominant-OR** bridging fault model, assumes that one driver dominates the logic value of the shorted nets for one logic value only [Stroud 2002].

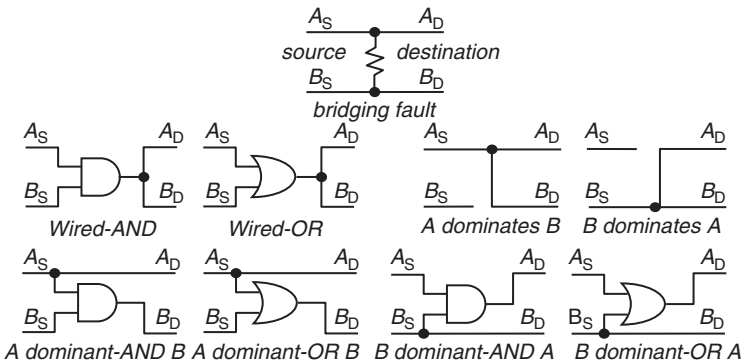



**FIGURE 1.9**

Bridging fault models.

4. **Delay fault models**: Resistive opens and shorts in wires and parameter variations in transistors can cause excessive delay such that the total propagation delay falls outside the specified limit. **Delay faults** have become more prevalent with decreasing feature sizes, and there are different delay fault models. In **gate-delay fault** and **transition fault** models, a delay fault occurs when the time interval taken for a transition through a single gate exceeds its specified range. The **path-delay fault** model, on the other hand, considers the cumulative propagation delay along any signal path through the circuit. The **small delay defect** model takes timing delay associated with the fault sites and propagation paths from the layout into consideration [Sato 2005; Wang 2007].

### 1.3.2 Design for testability

To test a given circuit, we need to control and observe logic values of internal nodes. Unfortunately, some nodes in sequential circuits can be difficult to control and observe. DFT techniques have been proposed to improve the controllability and observability of internal nodes and generally fall into one of the following three categories: (1) **ad-hoc DFT** methods, (2) **scan design**, and (3) ***built-in self-test*** (BIST). Ad-hoc methods were the first DFT technique introduced in the 1970s [Abramovici 1994]. The goal was to target only portions of the circuit that were difficult to test and to add circuitry (typically **test point** insertion) to improve the controllability and/or observability of internal nodes [Wang 2006].

**Scan design** was the most significant DFT technique proposed [Williams 1983]. This is because the scan design implementation process was easily automated and incorporated in the EDA flow. A scan design can be flip-flop based or latch based. The latch-based scan design is commonly referred to as ***level-sensitive scan design*** (LSSD) [Eichelberger 1978]. The basic idea to create a scan design is to reconfigure each flip-flop (*FF*) or latch in the sequential circuit to become a ***scan flip-flop*** (*SFF*) or **scan latch** (often called **scan cell**), respectively. These scan cells, as illustrated in Figure 1.10, are then connected in series to form a shift register, or **scan chain**, with direct access to a primary input (*Scan Data In*) and a primary output (*Scan Data Out*). During the shift operation (when *Scan Mode* is set to 1), the scan chain is used to shift in a test pattern from *Scan Data In* to be applied to the combinational logic. During one clock cycle of the normal system operation (when *Scan Mode* is set to 0), the test pattern is applied to the combinational logic and the output response is clocked back or captured into the scan cells. The scan chain is then used in scan mode to shift out the combinational logic output response while shifting in the next test pattern to be applied. As a result, scan design reduces the problem of testing sequential logic to that of testing combinational logic and, thereby, facilitates the use of *automatic test pattern generation* (ATPG) techniques and software developed for combinational logic.

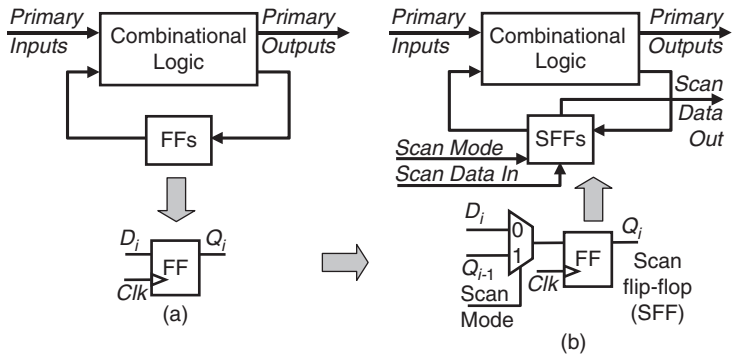



**FIGURE 1.10**

Transforming a sequential circuit to flip-flop-based scan design: (a) Example of a sequential circuit. (b) Example of a scan design.

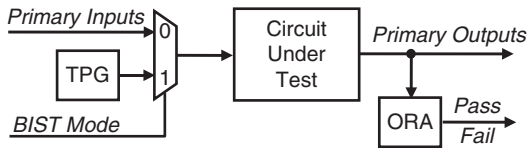



**FIGURE 1.11**

Simple BIST architecture.

BIST was proposed around 1980 to embed test circuitry in the device or system to perform self-test internally. As illustrated in Figure 1.11, a ***test pattern generator*** (TPG) is used to automatically supply the internally generated test patterns to the ***circuit under test*** (CUT), and an ***output response analyzer*** (ORA) is used to compact the output responses from the CUT [Stroud 2002]. Because the test circuitry resides with the CUT, BIST can be used at all levels of testing from wafer through system level testing. BIST is typically applied on the basis of the type of circuit under test. For example, scan-based BIST approaches are commonly used for general sequential logic (often called **logic BIST**); more algorithmic BIST approaches are used for regular structures such as memories (often called **memory BIST**). Because of the complexity of current VLSI devices that can include ***analog and mixed-signal*** (AMS) circuits, as well as hundreds of memories, BIST implementations are becoming an essential part of both system and test requirements [Wang 2006, 2007].

Test compression can be considered as a supplement to scan design and is commonly used to reduce the amount of test data (both input stimuli and output responses) that must be stored on the ***automatic test equipment*** (ATE) [Touba 2006]. Reduction in test data volume and test application time by 10× or more can be achieved. This is typically done by including a decompressor before the $m$ scan chain inputs of the CUT to decompress the compressed input

stimuli and a compactor after the $m$ scan chain outputs of the CUT to compact output responses, as illustrated in Figure 1.12. The compressed input stimulus and compacted output response are each connected to $n$ tester channels on the ATE, where $n < m$ and $n$ is typically at least 10× smaller than $m$. Modern test synthesis tools can now directly incorporate these test compression features into either an RTL design or a gate-level design as will be discussed in more detail in Chapter 3.

### 1.3.3 **Fault simulation and test generation**

The mechanics of testing for fault simulation, as illustrated in Figure 1.13, are similar at all levels of testing, including design verification. First, a set of target faults (fault list) based on the CUT is enumerated. Often, **fault collapsing** is applied to the enumerated fault set to produce a collapsed fault set to reduce fault simulation or fault grading time. Then, input stimuli are applied to the CUT, and the output responses are compared with the expected fault-free responses to determine whether the circuit is faulty. For fault simulation, the CUT is typically synthesized down to a gate-level design (or circuit netlist).

Ensuring that sufficient design verification has been obtained is a difficult step for the designer. Although the ultimate determination is whether or not the design works in the system, fault simulation, illustrated in Figure 1.13, can provide a rough quantitative measure of the level of design verification much earlier in the design process. Fault simulation also provides valuable information

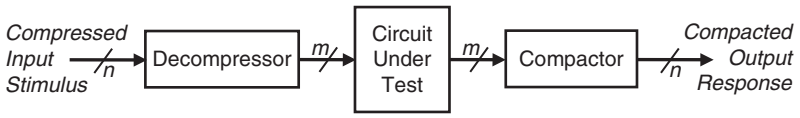



**FIGURE 1.12**

Test compression architecture.

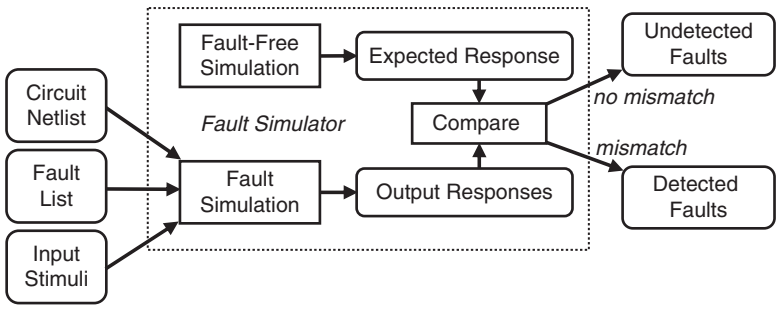



**FIGURE 1.13**

Fault simulation.

on portions of the design that need further design verification, because design verification vectors are often used as functional vectors (called **functional testing**) during manufacturing test.

Test development consists of selecting specific test patterns on the basis of circuit structural information and a set of **fault models**. This approach, called **structural testing**, saves test time and improves test efficiency, because the total number of test patterns is largely decreased since the test patterns target specific faults that would result from defects in the manufactured circuit. Structural testing cannot guarantee detection of all possible manufacturing defects, because the test patterns are generated on the basis of specific fault models. However, fault models provide a quantitative measure of the fault detection capabilities for a given set of test patterns for the targeted fault model; this measure is called **fault coverage** and is defined as:

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

Any input pattern, or sequence of input patterns, that produces a different output response in a faulty circuit from that of the fault-free circuit is a test pattern, or sequence of test patterns, which will detect the fault. Therefore, the goal of ***automatic test pattern generation*** (ATPG) is to find a set of test patterns that detects all faults considered for that circuit. Because a given set of test patterns is usually capable of detecting many faults in a circuit, **fault simulation** is typically used to evaluate the fault coverage obtained by that set of test patterns. As a result, fault models are needed for fault simulation and for ATPG.

### 1.3.4 Manufacturing test

The tester, also referred to as the ***automatic test equipment*** (ATE), applies the functional test vectors and structural test patterns to the fabricated circuit and compares the output responses with the expected responses obtained from the design verification simulation environment for the fault-free (and hopefully, design error-free) circuit. A "faulty" circuit is now considered to be a circuit with manufacturing defects.

Some percentage of the manufactured devices, boards, and systems is expected to be faulty because of manufacturing defects. As a result, testing is required during the manufacturing process in an effort to find and eliminate those defective parts. The **yield** of a manufacturing process is defined as the percentage of acceptable parts among all parts that are fabricated:

$$yield = \frac{number\ of\ acceptable\ parts}{total\ number\ of\ parts\ fabricated}$$

A **fault** is a representation of a defect reflecting a physical condition that causes a circuit to fail to perform in a required manner. When devices or electronic systems are tested, the following two undesirable situations may occur: (1) a faulty circuit appears to be a good part passing the test, or (2) a good circuit fails the test and appears as faulty. These two outcomes are often due to a poorly designed test or the lack of DFT. As a result of the first case, even if all products pass the manufacturing test, some faulty devices will still be found in the manufactured electronic system. When these faulty circuits are returned to the manufacturer, they undergo *failure mode analysis* (FMA) or fault diagnosis for possible improvements to the manufacturing process [Wang 2006]. The ratio of field-rejected parts to all parts passing quality assurance testing is referred to as the **reject rate**, also called the **defect level**:

$$reject\ rate = \frac{number\ of\ faulty\ parts\ passing\ final\ test}{total\ number\ of\ parts\ passing\ final\ test}$$

Because of unavoidable statistical flaws in the materials and masks used to fabricate the devices, it is impossible for 100% of any particular kind of device to be defect free. Thus, the first testing performed during the manufacturing process is to test the devices fabricated on the wafer to determine which devices are defective. The chips that pass the wafer-level test are extracted and packaged. The packaged devices are retested to eliminate those devices that may have been damaged during the packaging process or put into defective packages. Additional testing is used to ensure the final quality before shipping to customers. This final testing includes measurement of parameters such as input/output timing specifications, voltage, and current. In addition, **burn-in** or **stress testing** is often performed when chips are subject to high temperature and supply voltage. The purpose of burn-in testing is to accelerate the effect of defects that could lead to failures in the early stages of operation of the device. FMA is typically used at all stages of the manufacturing test to identify improvements to processes that will result in an increase in the number of defect-free electronic devices and systems produced.

In the case of a VLSI device, the chip may be discarded or it may be investigated by FMA for yield enhancement. In the case of a PCB, FMA may be performed for yield enhancement or the board may undergo further testing for fault location and repair. A "good" circuit is assumed to be defect free, but this assumption is only as good as the quality of the tests being applied to the manufactured design. Once again, fault simulation provides a quantitative measure of the quality of a given set of tests.

CHAPTER 3

# Design for testability

Laung-Terng (L.-T.) Wang
*SynTest Technologies, Inc., Sunnyvale, California*

## ABOUT THIS CHAPTER

***Design for testability*** (DFT) has become an essential part for designing ***very-large-scale integration*** (VLSI) circuits. The most popular DFT techniques in use today for testing the digital portion of the VLSI circuits include **scan** and **scan-based logic** ***built-in self-test*** (BIST). Both techniques have proved to be quite effective in producing testable VLSI designs. In addition, **test compression**, a supplemental DFT technique for scan, is growing in importance for further reduction in test data volume and test application time during manufacturing test.

To provide readers with an in-depth understanding of the most recent DFT advances in scan, logic BIST, and test compression, this chapter covers a number of fundamental DFT techniques to facilitate testing of modern digital circuits. These techniques are required to improve the product quality and reduce the defect level and test cost of a digital circuit, while at the same time simplifying the test, debug, and diagnosis tasks.

In this chapter, we first cover the basic DFT concepts and methods for performing testability analysis. Next, **scan design**, the most widely used structured DFT method, is discussed, including popular scan cell designs, scan architectures, and at-speed clocking schemes. After a brief introduction to the basic concept of logic BIST, we then discuss BIST pattern generation and output response analysis schemes along with a number of logic BIST architectures for in-circuit self-test. Finally, we present a number of test compression circuit structures for test stimuli compression and test response compaction. The chapter also includes a description of logic BIST and test compression architectures currently practiced in industry.

## 3.1 INTRODUCTION

With advances in semiconductor manufacturing technology, ***integrated circuits*** (ICs) can now contain tens to hundreds of millions of transistors running in the gigahertz range. The production and use of these integrated circuits has run into a variety of test challenges during wafer probe, wafer sort, preship screening, incoming test of chips and boards, test of assembled boards, system test, periodic maintenance, repair test, etc. During the early stages of IC production history, design and test were regarded as separate functions, performed by separate and unrelated groups of engineers. During these early years, a design engineer's job was to implement the required functionality on the basis of design specifications, without giving any thought to how the manufactured device was to be tested. Once the functionality was implemented, the design information was transferred to test engineers. A test engineer's job was to determine how to best test each manufactured device within a reasonable amount of time and to screen out the parts that may contain manufacturing defects while shipping all defect-free devices to customers. The final quality of the test was determined by keeping track of the number of defective parts shipped to the customers on the basis of customer returns. This product quality, measured in terms of ***defective parts per million*** (DPM) shipped, was a final test score for quantifying the effectiveness of the developed test.

Although this approach worked well for small-scale integrated circuits that mainly consisted of combinational logic or simple finite-state machines, it was unable to keep up with the circuit complexity as designs moved from ***small-scale integration*** (SSI) to ***very large-scale integration*** (VLSI). A common approach to testing these VLSI devices during the 1980s relied heavily on fault simulation to measure the fault coverage of the supplied functional patterns. Functional patterns were developed to navigate through the long sequential depths of a design, hoping to exercise all internal states and to detect all possible manufacturing defects. A **fault simulation** or **fault-grading** tool was used to quantify the effectiveness of the functional patterns. If the supplied functional patterns did not reach the target fault coverage goal, additional functional patterns were added. Unfortunately, this approach typically failed to improve the circuit's fault coverage beyond 80%, and the quality of the shipped products suffered.

Gradually, it became clear that designing devices without paying much attention to test resulted in increased test cost and decreased test quality. Some designs, which were otherwise best-in-class with regard to functionality and performance, failed commercially because of prohibitive test costs or poor product quality. These problems have since led to the development and deployment of DFT engineering in the industry.

The first challenge facing DFT engineers was to find simpler ways of exercising all internal states of a design and reaching the target fault coverage goal.

Various **testability measures** and **ad hoc testability enhancement** methods were proposed and used in the 1970s and 1980s to serve this purpose. These methods were mainly used to aid in the circuit's **testability** or to increase the circuit's **controllability** and **observability** [McCluskey 1986; Abramovici 1994]. Although attempts to use these methods have substantially improved the testability of a design and eased sequential ***automatic test pattern generation*** (ATPG), their end results at reaching the target fault coverage goal were far from satisfactory; it was still quite difficult to reach more than 90% fault coverage for large designs. This was mostly because even with these testability aids, deriving functional patterns by hand or generating test patterns for a sequential circuit is a much more difficult problem than generating test patterns for a combinational circuit [Fujiwara 1982; Bushnell 2000; Jha 2003].

Today, the semiconductor industry relies heavily on two techniques for testing digital circuits: *scan* and *logic built-in self-test* (BIST) [Abramovici 1994; McCluskey 1986]. **Scan** converts a digital sequential circuit into a scan design and then uses ATPG software [Bushnell 2000; Jha 2003; Wang 2006a] to detect faults that are caused by manufacturing defects (physical failures) and manifest themselves as errors, whereas logic BIST requires the use of a portion of the VLSI circuit to test itself on-chip, on-board, or in-system. To keep up with the design and test challenges [SIA 2005, 2006], more advanced ***design-for-testability*** (DFT) techniques, such as test compression, at-speed delay fault testing, and power-aware test generation, have been developed over the past few years to further address the test cost, delay fault, and test power issues [Gizopoulos 2006; Wang 2006a, 2007a].

**Scan design** is implemented by first replacing all selected storage elements of the digital circuit with **scan cells** and then connecting them into one or more shift registers, called **scan chains**, to provide them with external access. With external access, one can now control and observe the internal states of the digital circuit by simply shifting test stimuli into and test responses out of the shift registers during scan testing. The DFT technique has since proved to be quite effective in improving the product quality, testability, and diagnosability of scan designs [Crouch 1999; Bushnell 2000; Jha 2003; Gizopoulos 2006; Wang 2006a, 2007a]. Although scan has offered many benefits during manufacturing test, it is becoming inefficient to test deep submicron or nanometer VLSI designs. The reasons are mostly because (1) traditional test schemes that use ATPG software to target single faults have become quite expensive and (2) sufficiently high fault coverage for these deep submicron or nanometer VLSI designs is hard to sustain from the chip level to the board and system levels.

To alleviate these test problems, the scan approach is typically combined with **logic BIST** that incorporates BIST features into the scan design at the design stage [Bushnell 2000; Mourad 2000; Stroud 2002; Jha 2003]. With logic BIST, circuits that generate test patterns and analyze the output responses of the functional circuitry are embedded in the chip or elsewhere on the same board where the chip resides to test the digital logic circuit itself. Typically,

pseudo-random patterns are applied to the ***circuit under test*** (CUT), while their test responses are compacted in a ***multiple-input signature register*** (MISR) [Bardell 1987; Rajski 1998; Nadeau-Dostie 2000; Stroud 2002; Jha 2003; Wang 2006a]. Logic BIST is crucial in many applications, in particular, for safety-critical and mission-critical applications. These applications, commonly found in the aerospace/defense, automotive, banking, computer, healthcare, networking, and telecommunications industries, require on-chip, on-board, or in-system self-test to improve the reliability of the entire system, as well as the ability to perform in-field diagnosis.

Since the early 2000s, **test compression**, a supplemental DFT technique to scan, is gaining industry acceptance to further reduce test data volume and test application time [Touba 2006; Wang 2006a]. Test compression involves compressing the amount of test data (both test stimulus and test response) that must be stored on ***automatic test equipment*** (ATE) for testing with a deterministic ATPG-generated test set. This is done by use of **code-based schemes** or adding additional on-chip hardware before the scan chains to decompress the test stimulus coming from the ATE and after the scan chains to compress the test response going to the ATE. This differs from logic BIST in that the test stimuli that are applied to the CUT are a deterministic (ATPG-generated) test set rather than pseudo-random patterns. Typically, test compression can provide 10× to 100× or even more reduction in test application time and test data volume and hence can drastically save scan test cost.

## 3.2 TESTABILITY ANALYSIS

***Testability*** is a relative measure of the effort or cost of testing a logic circuit. In general, it is based on the assumption that only primary inputs and primary outputs can be directly controlled and observed, respectively. Testability reflects the effort required to perform the main test operations of controlling internal signals from primary inputs and observing internal signals at primary outputs. **Testability analysis** refers to the process of assessing the testability of a logic circuit by calculating a set of numeric measures for each signal in the circuit.

One important application of testability analysis is to assist in the decision-making process during test generation. For example, if during test generation, it is determined that the output of a certain AND gate must be set to 0, testability analysis can help decide which AND gate input is the easiest to set to 0. The conventional application is to identify areas of poor testability to guide testability enhancement, such as test point insertion, for improving the testability of the design. For this purpose, testability analysis is performed at various design stages so that testability problems can be identified and fixed as early as possible.

Since the 1970s, many testability analysis techniques have been proposed [Rutman 1972; Stephenson 1976; Breuer 1978; Grason 1979]. The ***Sandia***

***Controllability/Observability Analysis Program*** (SCOAP) [Goldstein 1979, 1980] was the first topology-based program that populated testability analysis applications. Enhancements based on SCOAP have also been developed and used to aid in test point selection [Wang 1984, 1985]. These methods perform testability analysis by calculating the ***controllability*** and ***observability*** of each signal line, where *controllability* reflects the difficulty of setting a signal line to a required logic value from primary inputs, and *observability* reflects the difficulty of propagating the logic value of the signal line to primary outputs.

Traditionally, gate-level topologic information of a circuit is used for testability analysis. Depending on a target application, deterministic and/or random testability measures are calculated. In general, **topology-based testability analysis**, such as SCOAP or probability-based testability analysis, is computationally efficient but can produce inaccurate results for circuits containing many reconvergent fanouts. **Simulation-based testability analysis**, on the other hand, can generate more accurate estimates by simulating the circuit behavior with deterministic, random, or pseudo-random test patterns, but may require a long simulation time.

In this section, we first describe the method for performing SCOAP testability analysis. Then, probability-based testability analysis and simulation-based testability analysis are discussed.

### 3.2.1 SCOAP testability analysis

The SCOAP testability analysis program [Goldstein 1979, 1980] calculates six numeric values for each signal *s* in a logic circuit:

- CC0(*s*): Combinational 0-controllability of *s*
- CC1(*s*): Combinational 1-controllability of *s*
- CO(*s*): Combinational observability of *s*
- SC0(*s*): Sequential 0-controllability of *s*
- SC1(*s*): Sequential 1-controllability of *s*
- SO(*s*): Sequential observability of *s*

Roughly speaking, the three combinational testability measures, CC0, CC1, and CO, are related to the number of signals that need to be manipulated to control or observe *s* from primary inputs or at primary outputs, whereas the three sequential testability measures, SC0, SC1, and SO, are related to the number of clock cycles required to control or observe *s* from primary inputs or at primary outputs [Bushnell 2000]. The values of controllability measures range between 1 and infinite, whereas the values of observability measures range between 0 and infinite. As a boundary condition, the CC0 and CC1 values of a primary input are set to 1, the SC0 and SC1 values of a primary input are set to 0, and the CO and SO values of a primary output are set to 0.

#### 3.2.1.1 *Combinational controllability and observability calculation*

The first step in SCOAP is to calculate the combinational controllability measures of all signals. This calculation is performed from primary inputs toward primary outputs in a breadth-first manner. More specifically, the circuit is leveled from primary inputs to primary outputs to assign a *level order* for each gate. The output controllability for each gate is then scheduled in *level order* after the controllability measures of all of its inputs have been calculated. The rules for combinational controllability calculation are summarized in Table 3.1, where a 1 is added to each rule to indicate that a signal passes through one more level of logic gate. From this table, we can see that CC0($s$) $\geq 1$ and CC1($s$) $\geq 1$ for any signal $s$. A larger CC0($s$) or CC1($s$) value implies that it is more difficult to control $s$ to 0 or 1 from primary inputs.

Once the combinational controllability measures of all signals are calculated, the combinational observability of each signal can be calculated. This calculation is also performed in a breadth-first manner while moving from primary outputs toward primary inputs. The rules for combinational observability calculation are summarized in Table 3.2, where a 1 is added to each rule to indicate that a signal passes through one more level of logic. From this table, we can see that CO($s$) $\geq 0$ for any signal $s$. A larger CO($s$) value implies that it is more difficult to observe $s$ at any primary output.

**Table 3.1** SCOAP Combinational Controllability Calculation Rules

| | **0-Controllability** (Primary Input, Output, Branch) | **1-Controllability** (Primary Input, Output, Branch) |
|---|---|---|
| Primary Input | 1 | 1 |
| AND | *min* {input 0-controllabilities} + 1 | Σ (input 1-controllabilities) + 1 |
| OR | Σ (input 0-controllabilities) + 1 | *min* {input 1-controllability} + 1 |
| NOT | Input 1-controllability + 1 | Input 0-controllability + 1 |
| NAND | Σ (input 1-controllabilities) + 1 | *min* {input 0-controllability} + 1 |
| NOR | *min* {input 1-controllability) + 1 | Σ (input 0-controllabilities) + 1 |
| BUFFER | Input 0-controllability + 1 | Input 1-controllability + 1 |
| XOR | *min* {CC1($a$) + CC1($b$), CC0($a$) + CC0($b$)} + 1 | *min* {CC1($a$) + CC0($b$), CC0($a$) + CC1($b$)} + 1 |
| XNOR | *min* {CC1($a$) + CC0($b$), CC0($a$) + CC1($b$)} + 1 | *min* {CC1($a$) + CC1($b$), CC0($a$) + CC0($b$)} + 1 |
| Branch | Stem 0-controllability | Stem 1-controllability |

*a, b*: inputs of an XOR or XNOR gate

**Table 3.2** SCOAP Combinational Observability Calculation Rules

| **Observability** (Primary Output, Input, Stem) | |
|---|---|
| Primary Output | 0 |
| AND/NAND | Σ (output observability, 1-controllabilities of other inputs) + 1 |
| OR/NOR | Σ (output observability, 0-controllabilities of other inputs) + 1 |
| NOT/BUFFER | Output observability + 1 |
| XOR/XNOR | *a*: Σ (output observability, *min* {CC0(*b*), CC1(*b*)}) + 1<br>*b*: Σ (output observability, *min* {CC0(*a*), CC1(*a*)}) + 1 |
| Stem | *min* {branch observabilities} |

*a, b*: inputs of an XOR or XNOR gate

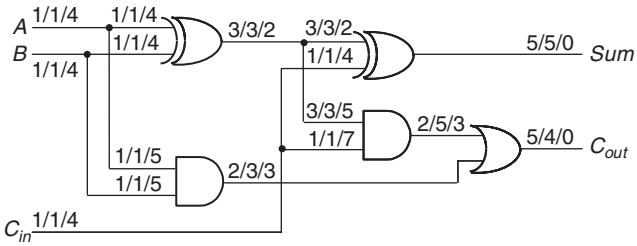



**FIGURE 3.1**

SCOAP full-adder example.

Figure 3.1 shows the combinational controllability and observability measures of a full-adder. The three-value tuple $v_1/v_2/v_3$ on each signal line represents the signal's 0-controllability ($v_1$), 1-controllability ($v_2$), and observability ($v_3$). The boundary condition is set by initializing the C0 and C1 values of the primary inputs *A, B,* and $C_{in}$ to 1, and the CO values of the primary outputs *Sum* and $C_{out}$ to 0. By applying the rules given in Tables 3.1 and 3.2 and starting with the given boundary condition, one can first calculate all combinational controllability measures forward and then calculate all combinational observability measures backward in *level order*.

### 3.2.1.2 *Sequential controllability and observability calculation*

Sequential controllability and observability measures are calculated in a similar manner as combinational measures, except that a 1 is not added as we move from one level of logic to another, but rather a 1 is added when a signal passes through a storage element. The difference is illustrated in the sequential circuit example shown in Figure 3.2, which consists of an AND gate and a positive

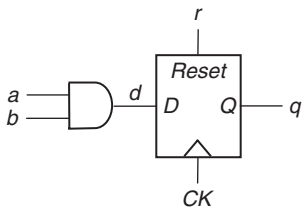



**FIGURE 3.2**

SCOAP sequential circuit example.

edge–triggered D flip-flop. The D flip-flop includes an active-high asynchronous reset pin *r*. SCOAP measures of a D flip-flop with a synchronous, as opposed to asynchronous, reset are shown in [Bushnell 2000].

First, we calculate the combinational and sequential controllability measures of all signals. To control signal *d* to 0, either input *a* or *b* must be set to 0. To control *d* to 1, both inputs *a* and *b* must be set to 1. Hence, the combinational and sequential controllability measures of signal *d* are:

$$CC0(d) = min\ \{CC0(a),\ CC0(b)\} + 1$$
$$SC0(d) = min\ \{SC0(a),\ SC0(b)\}$$
$$CC1(d) = CC1(a) + CC1(b) + 1$$
$$SC1(d) = SC1(a) + SC1(b)$$

To control the data output *q* of the D flip-flop to 0, the data input *d* and the reset signal *r* can be set to 0, while applying a rising clock edge (a 0-to-1 transition) to the clock *CK*. Alternately, this can be accomplished by setting *r* to 1 while holding *CK* at 0, without applying a clock pulse. Because a clock pulse is not applied to *CK*, a 1 is not added to the sequential controllability calculation in the second case. Therefore, the combinational and sequential 0-controllability measures of *q* are:

$$CC0(q) = min\{CC0(d) + CC0(CK) + CC1(CK) + CC0(r),\ CC1(r) + CC0(CK)\}$$
$$SC0(q) = min\{SC0(d) + SC0(CK) + SC1(CK) + SC0(r) + 1,\ SC1(r) + SC0(CK)\}$$

Here, CC0(*q*) measures how many signals in the circuit must be set to control *q* to 0, whereas SC0(*q*) measures how many flip-flops in the circuit must be clocked to set *q* to 0. To control the data output *q* of the D flip-flop to 1, the only way is to set the data input *d* to 1 and the reset signal *r* to 0, while applying a rising clock edge to the clock *CK*. Hence,

$$CC1(q) = CC1(d) + CC0(CK) + CC1(CK) + CC0(r)$$
$$SC1(q) = SC1(d) + SC0(CK) + SC1(CK) + SC0(r) + 1$$

Next, we calculate the combinational and sequential observability measures of all signals. The data input *d* can be observed at *q* by holding the reset signal *r* at 0 and applying a rising clock edge to *CK*. Hence,

$$CO(d) = CO(q) + CC0(CK) + CC1(CK) + CC0(r)$$
$$SO(d) = SO(q) + SC0(CK) + SC1(CK) + SC0(r) + 1$$

The asynchronous reset signal $r$ can be observed by first setting $q$ to 1, and then holding $CK$ at the inactive state 0. Again, a 1 is not added to the sequential controllability calculation because a clock pulse is not applied to $CK$:

$$CO(r) = CO(q) + CC1(q) + CC0(CK)$$
$$SO(r) = SO(q) + SC1(q) + SC0(CK)$$

There are two ways to indirectly observe the clock signal $CK$ at $q$: (1) set $q$ to 1, $r$ to 0, $d$ to 0, and apply a rising clock edge at $CK$, or (2) set both $q$ and $r$ to 0, $d$ to 1, and apply a rising clock edge at $CK$. Hence,

$$CO(CK) = CO(q) + CC0(CK) + CC1(CK) + CC0(r) + min\{CC0(d) + CC1(q),\ CC1(d) + CC0(q)\}$$
$$SO(CK) = SO(q) + SC0(CK) + SC1(CK) + SC0(r) + min\{SC0(d) + SC1(q),\ SC1(d) + SC0(q)\} + 1$$

To observe an input of the AND gate at $d$ requires setting the other input to 1. Therefore, the combinational and sequential observability measures for both inputs $a$ and $b$ are:

$$CO(a) = CO(d) + CC1(b) + 1$$
$$SO(a) = SO(d) + SC1(b)$$
$$CO(b) = CO(d) + CC1(a) + 1$$
$$SO(b) = SO(d) + SC1(a)$$

It is important to note that controllability and observability measures calculated with SCOAP are heuristics, and only approximate the actual testability of a logic circuit. When scan design is used, testability analysis can assume that all scan cells are directly controllable and observable. It was also shown in [Agrawal 1982] that SCOAP may overestimate testability measures for circuits containing many reconvergent fanouts. However, with the capability of performing testability analysis in an $O(n)$ computational complexity for $n$ signals in a circuit, SCOAP provides a quick estimate of the circuit's testability that can be used to guide testability enhancement and test generation.

### 3.2.2 **Probability-based testability analysis**

Topology-based testability analysis techniques, such as SCOAP, have been found to be extremely helpful in supporting test generation, which is a main topic of Chapter 14. These testability measures are able to analyze the **deterministic testability** of the logic circuit in advance and during the ATPG search process [Ivanov 1988]. On the other hand, in logic ***built-in self-test*** (BIST), which is the main topic of Section 3.4, random or pseudo-random test patterns are generated without specifically performing deterministic test pattern generation operations on any signal line. In this case, topology-based testability measures that use signal probability to analyze the **random testability** of the circuit can be used [Parker 1975; Savir 1984; Jain 1985; Seth 1985]. These measures are often referred to as **probability-based testability measures** or probability-based testability analysis techniques.

For example, given a random input pattern, one can calculate three measures for each signal $s$ in a combinational circuit as follows:

- C0($s$): Probability-based 0-controllability of $s$
- C1($s$): Probability-based 1-controllability of $s$
- O($s$): Probability-based observability of $s$

Here, C0($s$) and C1($s$) are the probability of controlling signal $s$ to 0 and 1 from primary inputs, respectively. O($s$) is the probability of observing signal $s$ at primary outputs. These three probabilities range between 0 and 1. As a boundary condition, the C0 and C1 probabilities of a primary input are typically set to 0.5, and the O probability of a primary output is set to 1. For each signal $s$ in the circuit, $C0(s) + C1(s) = 1$.

Many methods have been developed to calculate the probability-based testability measures. A simple method is given in the following, whose basic procedure is similar to the one used for calculating combinational testability measures in SCOAP, except that different calculation rules are used. The rules for probability-based controllability and observability calculation are summarized in Tables 3.3 and 3.4, respectively. In Table 3.3, $p_0$ is the initial 0-controllability chosen for a primary input, where $0 < p_0 < 1$.

Compared with SCOAP testability measures, where non-negative integers are used, probability-based testability measures range between 0 and 1. The smaller

**Table 3.3** Probability-Based Controllability Calculation Rules

| | **0-Controllability** (Primary Input, Output, Branch) | **1-Controllability** (Primary Input, Output, Branch) |
|---|---|---|
| Primary Input | $p_0$ | $p_1 = 1 - p_0$ |
| AND | 1 − (output 1-controllability) | Π(input 1-controllabilities) |
| OR | Π(input 0-controllabilities) | 1 − (output 0-controllability) |
| NOT | Input 1-controllability | Input 0-controllability |
| NAND | Π(input 1-controllabilities) | 1 − (output 0-controllability) |
| NOR | 1 − (output 1-controllability) | Π(input 0-controllabilities) |
| BUFFER | Input 0-controllability | Input 1-controllability |
| XOR | 1 − 1-controllability | Σ (C1($a$) × C0($b$),C0($a$) × C1($b$)) |
| XNOR | 1 − 1-controllability | Σ (C0($a$) × C0($b$),C1($a$) × C1($b$)) |
| Branch | Stem 0-controllability | Stem 1-controllability |

*a, b*: inputs of an XOR or XNOR gate

**Table 3.4** Probability-Based Observability Calculation Rules

| **Observability** (Primary Output, Input, Stem) | |
|---|---|
| Primary output | 1 |
| AND/NAND | $\Pi$ (output observability, 1-controllabilities of other inputs) |
| OR/NOR | $\Pi$ (output observability, 0-controllabilities of other inputs) |
| NOT/BUFFER | Output observability |
| XOR/XNOR | *a*: $\Pi$ (output observability, *max* {0-controllability of *b*, 1-controllability of *b*})<br>*b*: $\Pi$ (output observability, *max* {0-controllability of *a*, 1-controllability of *a*}) |
| Stem | *max* {branch observabilities} |

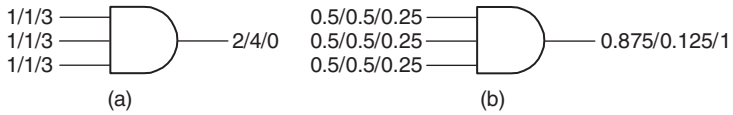



**FIGURE 3.3**

Comparison of SCOAP and probability-based testability measures: (a) SCOAP combinational measures. (b) Probability-based measures.

a probability-based testability measure of a signal, the more difficult it is to control or observe the signal. Figure 3.3 illustrates the difference between SCOAP testability measures and probability-based testability measures of a 3-input AND gate. The three-value tuple $v_1/v_2/v_3$ of each signal line represents the signal's 0-controllability ($v_1$), 1-controllability ($v_2$), and observability ($v_3$).

Signals with poor probability-based testability measures tend to be difficult to test with random or pseudo-random test patterns. The faults on these signal lines are often referred to as ***random pattern resistant*** (RP-resistant) [Savir 1984]. That is, either the probability of these signals randomly receiving a 0 or 1 from primary inputs, or the probability of observing these signals at primary outputs is low, assuming that all primary inputs have the equal probability of being set to 0 or 1.

The existence of such *RP-resistant faults* is the main reason why fault coverage that uses random or pseudo-random test patterns is low compared with the use of deterministic test patterns. In applications such as logic BIST, to solve this low fault coverage problem, test points are often inserted in the circuit to enhance the circuit's random testability. A few commonly used test point insertion techniques are discussed in [Wang 2006a].

### 3.2.3 Simulation-based testability analysis

In the calculation of SCOAP and probability-based testability measures as described previously, only the topologic information of a logic circuit is explicitly explored. These topology-based methods are static, in the sense that they do not use input test patterns for testability analysis. Their controllability and observability measures can be calculated in linear time, thus making them very attractive for applications that need fast testability analysis, such as test generation and logic BIST. However, the efficiency of these methods is achieved at the cost of reduced accuracy, especially for circuits that contain many reconvergent fanouts [Agrawal 1982].

As an alternative or supplement to static or topology-based testability analysis, dynamic or simulation-based methods that use input test patterns for testability analysis or testability enhancement can be performed through **statistical sampling**. Logic simulation and fault simulation techniques can be used [Bushnell 2000; Wang 2006a].

In statistical sampling, a sample set of input test patterns is selected, which is either generated randomly or derived from a given pattern set, and logic simulation is conducted to collect the responses of all or part of signal lines of interest. The commonly collected responses are the number of occurrences of 0's, 1's, 0-to-1 transitions, and 1-to-0 transitions, which are then used to profile statistically the testability of a logic circuit. These data are then analyzed to find locations of poor testability. If a signal line exhibits only a few transitions or no transitions for the sample input patterns, it might be an indication that the signal likely has poor controllability.

In addition to logic simulation, fault simulation has also been used to enhance the testability of a logic circuit with random or pseudo-random test patterns. For instance, a ***random resistant fault analysis*** (RRFA) method has been successfully applied to a high-performance microprocessor to improve the circuit's random testability in logic BIST [Rizzolo 2001]. This method is based on statistical data collected during fault simulation for a small number of random test patterns. Controllability and observability measures of each signal in the circuit are calculated by use of the probability models developed in the ***statistical fault analysis*** (STAFAN) algorithm [Jain 1985]. (STAFAN is the first method able to give reasonably accurate estimates of fault coverage in combinational circuits purely by use of input test patterns and without running fault simulation.) With these data, RRFA identifies signals that are difficult to control and/or observe, as well as signals that are statistically correlated. On the basis of the analysis results, RRFA then recommends test points to be added to the circuit to improve the circuit's random testability.

Because it can take a long simulation time to run through all input test patterns, these simulation-based methods are, in general, used to guide testability enhancement in test generation or logic BIST, when it is required to meet a very high fault coverage goal. This approach is crucial for life-critical and mission-critical applications, such as in the healthcare and defense/aerospace industries.

## 3.3 SCAN DESIGN

**Scan design** is currently the most widely used structured DFT approach. It is implemented by connecting selected storage elements of a design into one or more shift registers, called **scan chains**, to provide them with external access. Scan design accomplishes this task by replacing all selected storage elements with **scan cells**, each having one additional ***scan input*** (SI) port and one shared/additional ***scan output*** (SO) port. By connecting the SO port of one scan cell to the SI port of the next scan cell, one or more scan chains are created.

The scan-inserted design, called scan design, is now operated in three modes: **normal mode, shift mode**, and **capture mode**. Circuit operations with associated clock cycles conducted in these three modes are referred to as normal operation, shift operation, and capture operation, respectively.

In normal mode, all test signals are turned off, and the scan design operates in the original functional configuration. In both shift and capture modes, a **test mode** signal *TM* is often used to turn on all test-related fixes in compliance with scan design rules. A set of **scan design rules** that can be found in [Cheung 1997; Wang 2006a] are necessary to simplify the test, debug, and diagnose tasks, improve fault coverage, and guarantee the safe operation of the device under test. These circuit modes and operations are distinguished by use of additional test signals or test clocks. Fundamental scan architectures and at-speed clocking schemes are described in the following subsections.

### 3.3.1 Scan architectures

In this subsection, we first describe a few fundamental scan architectures. These fundamental scan architectures include (1) *muxed-D scan design*, in which storage elements are converted into muxed-D scan cells, (2) *clocked-scan design*, in which storage elements are converted into clocked-scan cells, and (3) *LSSD scan design*, in which storage elements are converted into *level-sensitive scan design* (LSSD) *shift register latches* (SRLs).

#### 3.3.1.1 *Muxed-D scan design*

Figure 3.4 shows a sequential circuit example with three D flip-flops. The corresponding muxed-D full-scan circuit is shown in Figure 3.5. An edge-triggered **muxed-D scan cell** design is shown in Figure 3.5a. This scan cell is composed of a D flip-flop and a multiplexer. The multiplexer uses a ***scan enable*** (*SE*) input to select between the ***data input*** (*DI*) and the ***scan input*** (*SI*).

In normal/capture mode, *SE* is set to 0. The value present at the data input *DI* is captured into the internal D flip-flop when a rising clock edge is applied. In shift mode, *SE* is set to 1. The scan input *SI* is now used to shift in new data to the D flip-flop, while the content of the D flip-flop is being shifted out. Sample operation waveforms are shown in Figure 3.5b. The three D flip-flops,

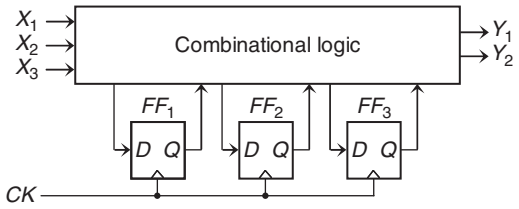



**FIGURE 3.4**

Sequential circuit example.

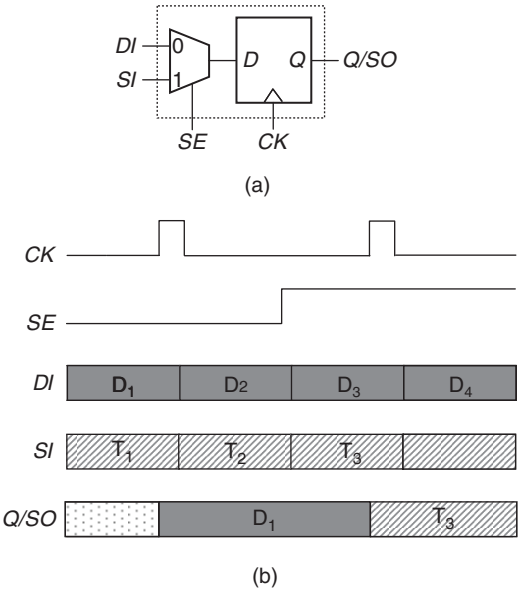



**FIGURE 3.5**

Edge-triggered muxed-D scan cell design and operation: (a) Muxed-D scan cell. (b) Sample waveforms.

$FF_1$, $FF_2$, and $FF_3$, shown in Figure 3.4, are replaced with three muxed-D scan cells, $SFF_1$, $SFF_2$, and $SFF_3$, respectively, shown in Figure 3.6.

In Figure 3.6, the data input *DI* of each scan cell is connected to the output of the combinational logic as in the original circuit. To form a scan chain, the scan inputs *SI* of $SFF_2$ and $SFF_3$ are connected to the outputs *Q* of the previous scan cells, $SFF_1$ and $SFF_2$, respectively. In addition, the scan input *SI* of the first scan cell $SFF_1$ is connected to the primary input *SI*, and the output *Q* of the last scan cell $SFF_3$ is connected to the primary output *SO*. Hence, in shift mode, *SE* is set to 1, and the scan cells operate as a single scan chain, which allows us to shift in any combination of logic values into the scan cells.

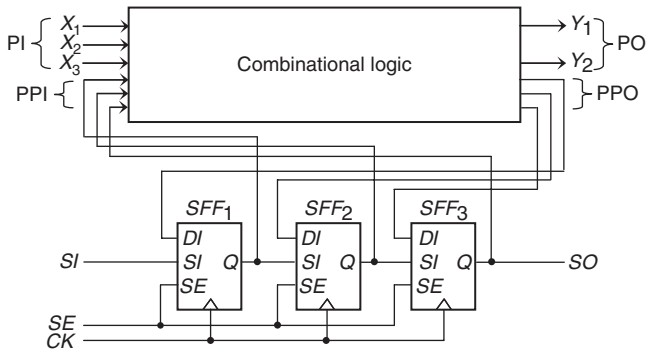



**FIGURE 3.6**

Muxed-D scan design.

In capture mode, *SE* is set to 0, and the scan cells are used to capture the test response from the combinational logic when a clock is applied.

In general, combinational logic in a full-scan circuit has two types of inputs: ***primary inputs*** (PIs) and ***pseudo primary inputs*** (PPIs). Primary inputs refer to the external inputs to the circuit, whereas pseudo primary inputs refer to the scan cell outputs. Both PIs and PPIs can be set to any required logic values. The only difference is that PIs are set directly in parallel from the external inputs, whereas PPIs are set serially through scan chain inputs. Similarly, the combinational logic in a full-scan circuit has two types of outputs: ***primary outputs*** (POs) and ***pseudo primary outputs*** (PPOs). Primary outputs refer to the external outputs of the circuit, whereas pseudo primary outputs refer to the scan cell inputs. Both POs and PPOs can be observed. The only difference is that POs are observed directly in parallel from the external outputs, whereas PPOs are observed serially through scan chain outputs.

#### 3.3.1.2 *Clocked-scan design*

An edge-triggered ***clocked-scan cell*** can also be used to replace a D flip-flop in a scan design [McCluskey 1986]. Similar to a muxed-D scan cell, a clocked-scan cell also has a data input *DI* and a scan input *SI*; however, in the clocked-scan cell, input selection is conducted with two independent clocks, data clock *DCK* and shift clock *SCK*, as shown in Figure 3.7a.

In normal/capture mode, the data clock *DCK* is used to capture the contents present at the data input *DI* into the clocked-scan cell. In shift mode, the shift clock *SCK* is used to shift in new data from the scan input *SI* into the clocked-scan cell, while the content of the clocked-scan cell is being shifted out. Sample operation waveforms are shown in Figure 3.7b.

The major advantage of the use of a clocked-scan cell is that it results in no performance degradation on the data input. A major disadvantage, however, is that it requires additional shift clock routing.

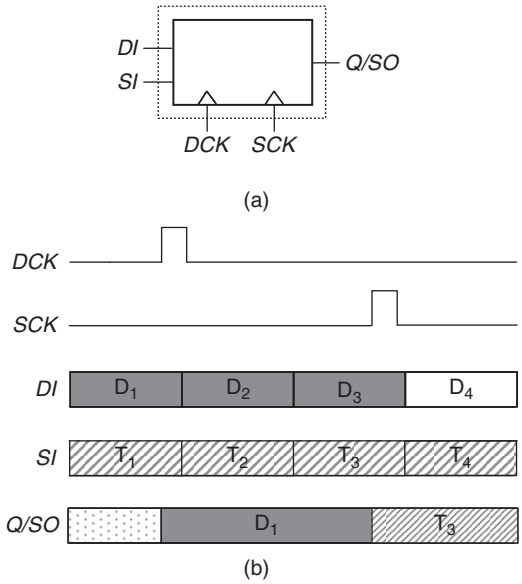



**FIGURE 3.7**

Clock-scan cell design and operation: (a) Clocked-scan cell. (b) Sample waveforms.

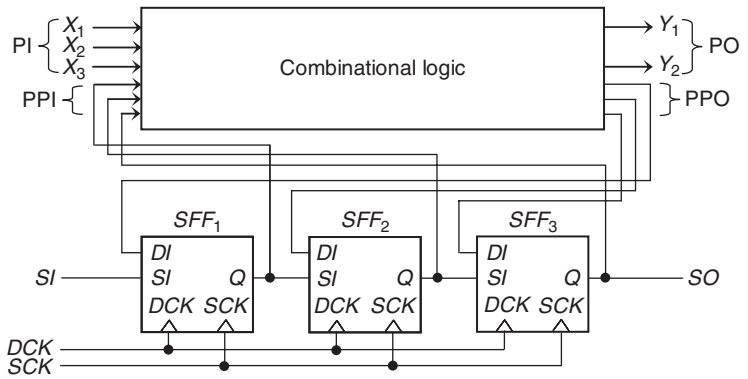



**FIGURE 3.8**

Clocked-scan design.

Figure 3.8 shows a clocked-scan design of the sequential circuit given in Figure 3.4. This clocked-scan design is tested with shift and capture operations, similar to a muxed-D scan design. The main difference is how these two operations are distinguished. In a muxed-D scan design, a scan enable signal *SE* is

used, as shown in Figure 3.6. In the clocked scan shown in Figure 3.8, these two operations are distinguished by properly applying the two independent clocks *SCK* and *DCK* during shift mode and capture mode, respectively.

### 3.3.1.3 *LSSD scan design*

Figure 3.9a shows a polarity-hold ***shift register latch*** (SRL) design described in [Eichelberger 1977] that can be used as an LSSD scan cell. This scan cell contains two latches, a master two-port D latch $L_1$ and a slave D latch $L_2$. Clocks *C, A*, and *B* are used to select between the data input *D* and the scan input *I* to drive $+L_1$ and $+L_2$.

To guarantee race-free operation, clocks *A, B*, and *C* are applied in a nonoverlapping manner. In designs in which $+L_1$ is used to drive the combinational logic, the master latch $L_1$ uses the system clock *C* to latch system data from the data input *D* and to output this data onto $+L_1$. In designs in which $+L_2$ is

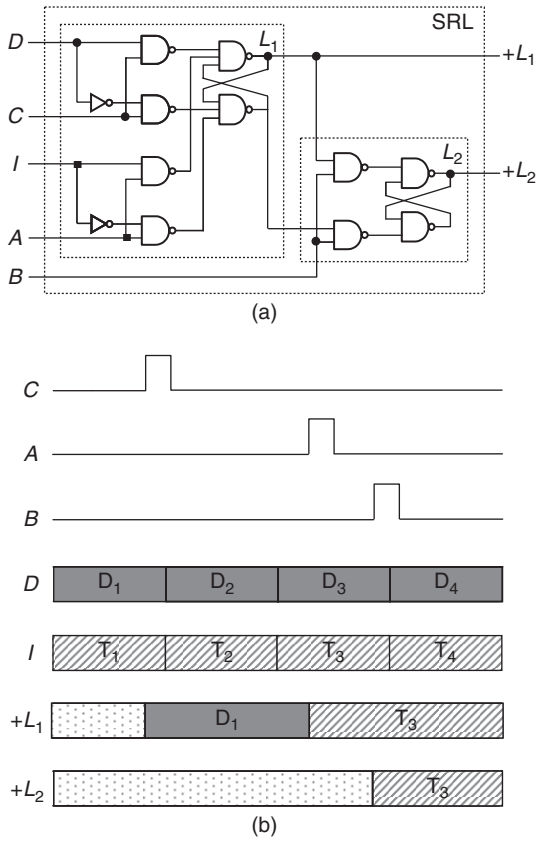



**FIGURE 3.9**

Polarity-hold SRL design and operation: (a) Polarity-hold SRL. (b) Sample waveforms.

used to drive the combinational logic, clock $B$ is used after clock $C$ to latch the system data from latch $L_1$ and to output these data onto $+L_2$. In both cases, capture mode uses both clocks $C$ and $B$ to output system data onto $+L_2$. Finally, in shift mode, clocks $A$ and $B$ are used to latch scan data from the scan input $I$ and to output these data onto $+L_1$, and then latch the scan data from latch $L_1$ and to output these data onto $+L_2$, which is then used to drive the scan input of the next scan cell. Sample operation waveforms are shown in Figure 3.9b.

LSSD scan designs can be implemented with either a **single-latch design** or a **double-latch design**. In single-latch design [Eichelberger 1977], the output port $+L_1$ of the master latch $L_1$ is used to drive the combinational logic of the design. In this case, the slave latch $L_2$ is used only for scan testing. Because LSSD designs use latches instead of flip-flops, at least two system clocks $C_1$ and $C_2$ are required to prevent combinational feedback loops from occurring. In this case, combinational logic driven by the master latches of the first system clock $C_1$ are used to drive the master latches of the second system clock $C_2$, and vice versa. For this to work, the system clocks $C_1$ and $C_2$ should be applied in a nonoverlapping fashion. Figure 3.10a shows an LSSD single-latch design with the polarity-hold SRL shown in Figure 3.9.

Figure 3.10b shows an example of LSSD **double-latch design** [DasGupta 1982]. In normal mode, the $C_1$ and $C_2$ clocks are used in a nonoverlapping manner, where the $C_2$ clock is the same as the $B$ clock. The testing of an LSSD scan design is conducted with shift and capture operations, similar to a muxed-D scan design. The main difference is how these two operations are distinguished. In a muxed-D scan design, a scan enable signal *SE* is used, as shown in Figure 3.6. In an LSSD scan design, these two operations are distinguished by properly applying nonoverlapping clock pulses to clocks $C_1$, $C_2$, $A$, and $B$. During the shift operation, clocks $A$ and $B$ are applied in a nonoverlapping manner, and the scan cells $SRL_1 \sim SRL_3$ form a single scan chain from *SI* to *SO*. During the capture operation, clocks $C_1$ and $C_2$ are applied in a nonoverlapping manner to load the test response from the combinational logic into the scan cells.

The major advantage of the use of an LSSD scan cell is that it allows us to insert scan into a latch-based design. In addition, designs that use LSSD are guaranteed to be race-free, which is not the case for muxed-D scan and clocked-scan designs. A major disadvantage, however, is that it requires routing for the additional clocks, which increases routing complexity.

The operation of a polarity-hold SRL is race-free if clocks $C$ and $B$, as well as $A$ and $B$, are nonoverlapping. This characteristic is used to implement LSSD circuits that are guaranteed to have race-free operation in normal mode and in test mode.

### 3.3.2 At-speed testing

Although scan design is commonly used in the industry for slow-speed stuck-at fault testing, its real value is in providing **at-speed testing** for high-speed and

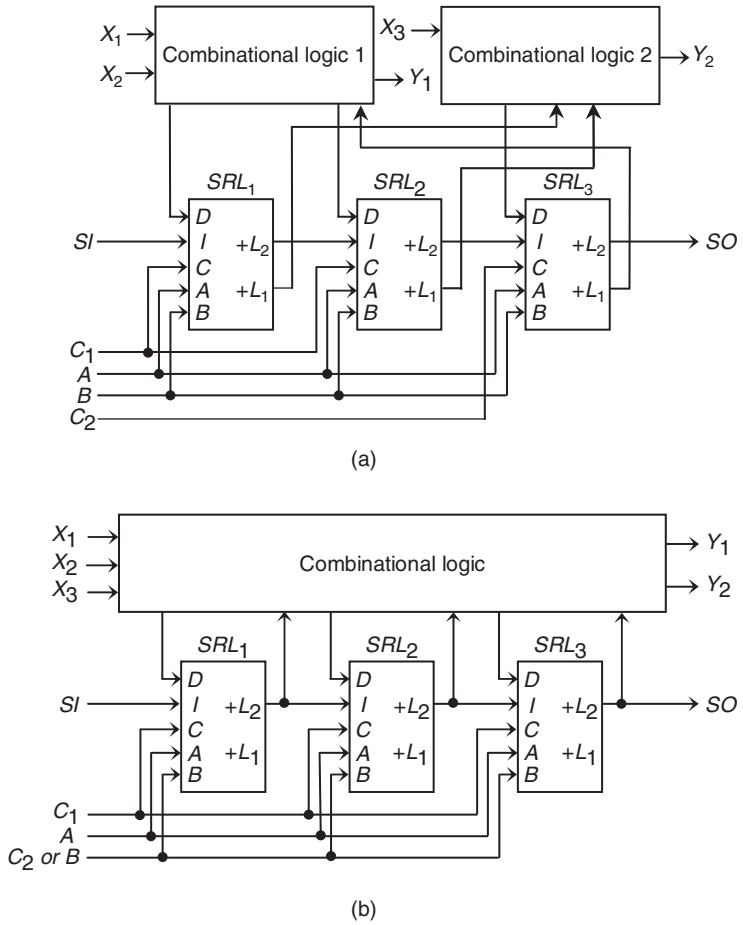



**FIGURE 3.10**

LSSD designs: (a) LSSD single-latch design. (b) LSSD double-latch design.

high-performance circuits. These circuits often contain multiple clock domains, each running at an operating frequency that is either synchronous or asynchronous to the other clock domains. Two clock domains are said to be **synchronous** if the active edges of both clocks controlling the two clock domains can be aligned precisely or triggered simultaneously. Two clock domains are said to be **asynchronous** if they are not synchronous.

There are two basic capture-clocking schemes for testing multiple clock domains at-speed: (1) *skewed-load* [Savir 1993] (also called *launch-on-shift* [LOS]) and (2) *double-capture* [Wang 2006a] (also called *launch-on-capture* [LOC] or *broad-side* [Savir 1994]). Both schemes can be used to test path-delay faults and transition faults within each clock domain (called **intra-clock-domain**

**faults**) or across clock domains (called **inter-clock-domain faults**). Skewed-load uses the last shift clock pulse followed immediately by a capture clock pulse to launch the transition and capture the output test response, respectively. Double-capture uses two consecutive capture clock pulses to launch the transition and capture the output test response, respectively. In both schemes, both launch and capture clock pulses must be running at the domain's operating speed or at-speed. The difference is that skewed-load requires the domain's scan enable signal *SE* to switch its value between the launch and capture clock pulses making *SE* act as a clock signal. Figure 3.11 shows sample waveforms that use the basic skewed-load and double-capture at-speed test schemes.

Scan designs typically include a few clock domains that will interact with one another. To guarantee the success of the capture operation, additional care must be taken in terms of the way the capture clocks are applied. This is mainly because the clock skew between different clock domains is typically large. To prevent this from happening, clocks can be applied sequentially (with the **staggered clocking** scheme [Wang 2005a, 2007b]), such that any clock skew that exists between the clock domains can be tolerated during the test generation process. It is also possible to apply only one clock during each capture operation by use of the **one-hot clocking** scheme. Most modern ATPG programs used currently can also automatically mask off unknown values (*X*'s) at the originating scan cells or receiving scan cells across clock domains. In this case, all clocks can also be applied simultaneously with the **simultaneous clocking** scheme [Wang 2007b]. During simultaneous clocking, if the launch clock pulses [Rajski 2003; Wang 2006a] or the capture clock pulses [Nadeau-Dostie 1994; Wang 2006a] can be aligned precisely, which applies only for synchronous clock domains, then the **aligned clocking** scheme can be used, and there is no need to mask off unknown values across these synchronous clock domains. These clocking schemes are illustrated in Figure 3.12.

In general, one-hot clocking produces the highest fault coverage at the expense of generating many more test patterns than other schemes. Simultaneous clocking can generate the smallest number of test patterns but may result

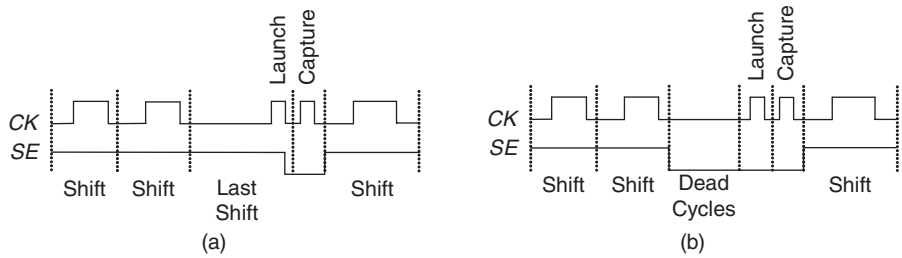



**FIGURE 3.11**

Basic at-speed test schemes: (a) Skewed-load. (b) Double-capture.

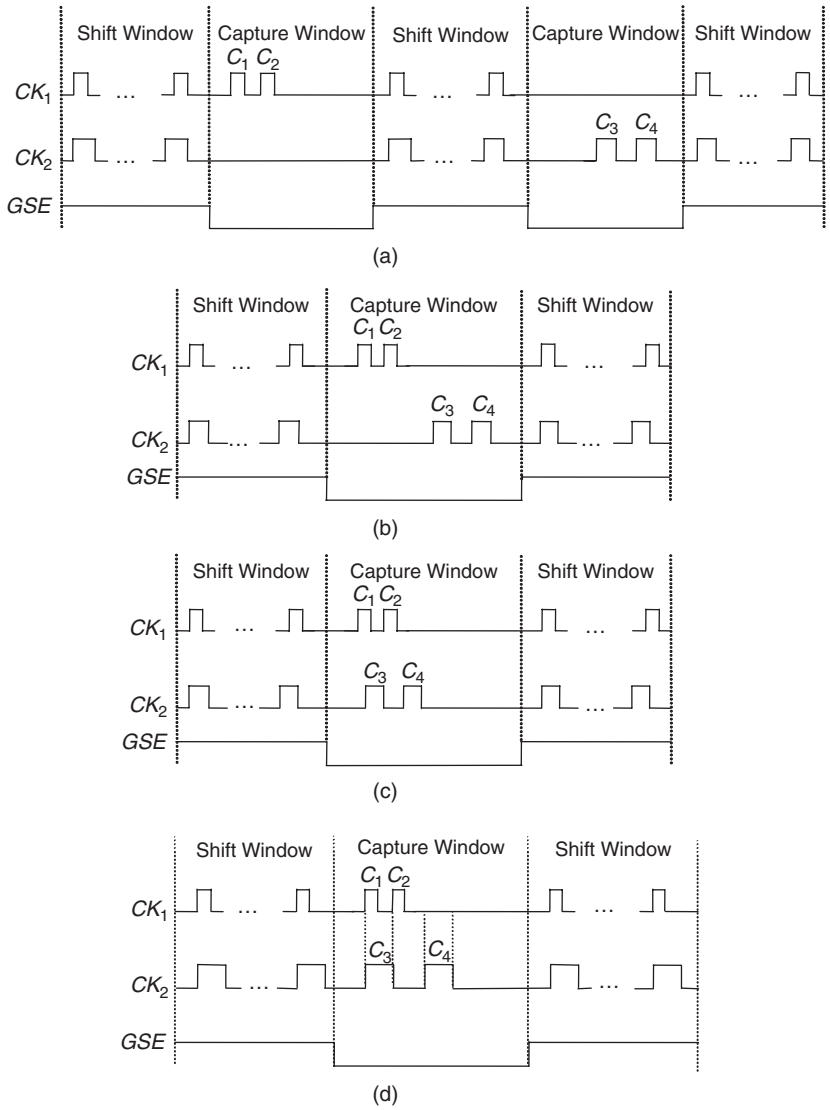



**FIGURE 3.12**

At-speed clocking schemes for testing two interacting clock domains: (a) One-hot clocking. (b) Staggered clocking. (c) Simultaneous clocking. (d) Aligned clocking.

in high fault coverage loss because of unknown ($X$) masking. The staggered clocking scheme is a happy medium because of its ability to generate test pattern count close to simultaneous clocking and fault coverage close to one-hot clocking. For large designs, it is no longer uncommon for transition fault ATPG

to take more than 2 to 4 weeks to complete. To reduce test generation time while at the same time obtaining the highest fault coverage, modern ATPG programs tend to either (1) run simultaneous clocking followed by one-hot clocking or (2) use staggered clocking followed by one-hot clocking. As a result, modern **at-speed scan architectures** now start supporting a combination of at-speed clocking schemes for test circuits comprising multiple synchronous and asynchronous clock domains. Some programs can even generate test patterns by mixing skewed-load and double-capture schemes.

## 3.4 LOGIC BUILT-IN SELF-TEST

*Logic built-in self-test* (BIST) requires using a portion of the circuit to test itself on-chip, on-board, or in-system. A typical logic BIST system is illustrated in Figure 3.13. The ***test pattern generator*** (TPG) automatically generates test patterns for application to the inputs of the ***circuit under test*** (CUT). The ***output response analyzer*** (ORA) automatically compacts the output responses of the CUT into a *signature*. Specific BIST timing control signals, including scan enable signals and clocks, are generated by the **logic BIST controller** for coordinating the BIST operation among the TPG, CUT, and ORA. The logic BIST controller provides a pass/fail indication once the BIST operation is complete. It includes comparison logic to compare the *final signature* with an embedded *golden signature*, and often comprises **diagnostic logic** for fault diagnosis. Because compaction is commonly used for output response analysis, it is required that all storage elements in the TPG, CUT, and ORA be initialized to known states before self-test, and no unknown ($X$) values are allowed to propagate from the CUT to the ORA. In other words, the CUT must comply with more stringent **BIST-specific design rules** [Wang 2006a] in addition to those *scan design rules* required for scan design.

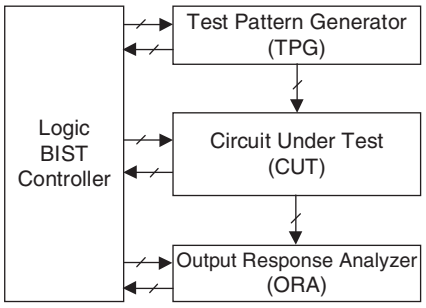



**FIGURE 3.13**

A typical logic BIST system.

### 3.4.1 **Test pattern generation**

For logic BIST applications, in-circuit TPGs constructed from ***linear feedback shift registers*** **(LFSRs)** are most commonly used to generate test patterns or test sequences for exhaustive testing, pseudo-random testing, and pseudo-exhaustive testing.

**Exhaustive testing** always guarantees 100% single-stuck and multiple-stuck fault coverage. This technique requires all possible $2^n$ test patterns to be applied to an $n$-input combinational CUT, which can take too long for combinational circuits where $n$ is huge. Therefore, **pseudo-random testing** [Bardell 1987] is often used for generating a subset of the $2^n$ test patterns and uses fault simulation to calculate the exact fault coverage. In some cases, this might become quite time-consuming, if not infeasible. To eliminate the need for fault simulation while at the same time maintaining 100% single-stuck fault coverage, we can use **pseudo-exhaustive testing** [McCluskey 1986] to generate $2^w$ or $2^k - 1$ test patterns, where $w < k < n$, when each output of the $n$-input combinational CUT at most depends on $w$ inputs. For testing delay faults, hazards must also be taken into consideration.

**Standard LFSR**

Figure 3.14 shows an $n$-stage **standard LFSR**. It consists of $n$ D flip-flops and a selected number of *exclusive-OR* (XOR) gates. Because XOR gates are placed on the external feedback path, the standard LFSR is also referred to as an **external-XOR LFSR** [Golomb 1982].

**Modular LFSR**

Similarly, an $n$-stage **modular LFSR** with each XOR gate placed between two adjacent D flip-flops, as shown in Figure 3.15, is referred to as an **internal-XOR LFSR** [Golomb 1982]. The modular LFSR runs faster than its corresponding standard LFSR, because each stage introduces at most one XOR-gate delay.

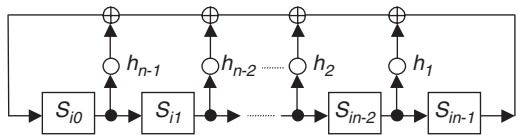



**FIGURE 3.14**

An $n$-stage (external-XOR) standard LFSR.

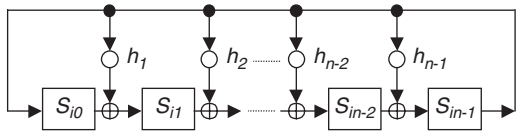



**FIGURE 3.15**

An $n$-stage (internal-XOR) modular LFSR.

**LFSR Properties**

The internal structure of the $n$-stage LFSR in each figure can be described by specifying a **characteristic polynomial** of degree $n$, $f(x)$, in which the symbol $h_i$ is either 1 or 0, depending on the existence or absence of the feedback path, where

$$f(x) = 1 + h_1x + h_2x^2 + \ldots + h_{n-1}x^{n-1} + x^n$$

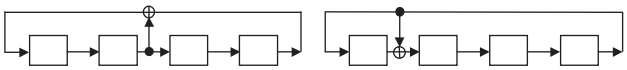
Let $S_i$ represent the contents of the $n$-stage LFSR after $i$th shifts of the initial contents, $S_0$, of the LFSR, and $S_i(x)$ be the polynomial representation of $S_i$. Then, $S_i(x)$ is a polynomial of degree $n-1$, where

$$S_i(x) = S_{i0} + S_{i1}x + S_{i2}x^2 + \ldots + S_{in\text{-}2}x^{n-2} + S_{in-1}x^{n-1}$$

If $T$ is the smallest positive integer such that $f(x)$ divides $1 + x^T$, then the integer $T$ is called the **period** of the LFSR. If $T = 2^n-1$, then the $n$-stage LFSR generating the **maximum-length sequence** is called a **maximum-length LFSR**.

For example, consider the four-stage standard and modular LFSRs shown in Figures 3.16a and 3.16b below. The characteristic polynomials, $f(x)$, used to construct both LFSRs are $1 + x^2 + x^4$ and $1 + x + x^4$, respectively.

The test sequences generated by each LFSR, when its initial contents, $S_0$, are set to {0001} or $S_0(x) = x^3$, are listed in Figures 3.16c and 3.16d, respectively.



(a) (b)

| (c) | (d) |
|---|---|
| **0 0 0 1** | **0 0 0 1** |
| 1 0 0 0 | 1 1 0 0 |
| 0 1 0 1 | 0 1 1 0 |
| 1 0 1 0 | 0 0 1 1 |
| 0 1 0 1 | 1 1 0 1 |
| 0 0 1 0 | 1 0 1 0 |
| **0 0 0 1** | 0 1 0 1 |
| 1 0 0 0 | 1 1 1 0 |
| 0 1 0 1 | 0 1 1 1 |
| 1 0 1 0 | 1 1 1 1 |
| 0 1 0 1 | 1 0 1 1 |
| 0 0 1 0 | 1 0 0 1 |
| **0 0 0 1** | 1 0 0 0 |
| 1 0 0 0 | 0 1 0 0 |
| 0 1 0 0 | 0 0 1 0 |
| 1 0 1 0 | **0 0 0 1** |

**FIGURE 3.16**

Example four-stage test pattern generators (TPGs): (a) Four-stage standard LFSR. (b) Four-stage modular LFSR. (c) Test sequence generated by (a). (d) Test sequence generated by (b).

Because the first test sequence repeats after 6 patterns and the second test sequence repeats after 15 patterns, the LFSRs have periods of 6 and 15, respectively. This further implies that $1 + x^6$ can be divided by $1 + x^2 + x^4$, and $1 + x^{15}$ can be divided by $1 + x + x^4$.

Define a **primitive polynomial** of degree $n$ over **Galois field** $GF(2)$, $p(x)$, as a polynomial that divides $1 + x^T$, but not $1 + x^i$, for any integer $i < T$, where $T = 2^n - 1$ [Golomb 1982]. A primitive polynomial is **irreducible**. Because $T = 15 = 2^4 - 1$, the characteristic polynomial, $f(x) = 1 + x + x^4$, used to construct Figure 3.16b is a primitive polynomial, and thus the modular LFSR is a maximum-length LFSR. Let

$$r(x) = f(x)^{-1} = x^n f(x^{-1})$$

Then $r(x)$ is defined as a **reciprocal polynomial** of $f(x)$ [Peterson 1972]. A reciprocal polynomial of a primitive polynomial is also a primitive polynomial. Thus, the reciprocal polynomial of $f(x) = 1 + x + x^4$ is also a primitive polynomial, with $p(x) = r(x) = 1 + x^3 + x^4$.

Table 3.5 lists a set of primitive polynomials of degree $n$ up to 100. It was taken from [Bardell 1987]. A different set was given in [Wang 1988a]. Each polynomial can be used to construct minimum-length LFSRs in standard or modular form. For primitive polynomials of degree up to 300, consult [Bardell 1987].

#### 3.4.1.1 *Exhaustive testing*

**Exhaustive testing** requires applying $2^n$ exhaustive patterns to an $n$-input combinational CUT. Any **binary counter** can be used as an ***exhaustive pattern generator*** (EPG) for this purpose. Figure 3.17 shows an example of a 4-bit binary counter design for testing a 4-input combinational CUT.

Exhaustive testing guarantees that all detectable, combinational faults (those that do not change a combinational circuit into a sequential circuit) will be detected. This approach is especially useful for circuits in which the number of inputs, $n$, is a small number (*e.g.*, 20 or less). When $n$ is larger than 20, the test time may be prohibitively long and is thus not recommended. The following techniques are aimed at reducing the number of test patterns. They are recommended when exhaustive testing is impractical.

#### 3.4.1.2 *Pseudo-random testing*

One approach, which can reduce test length but sacrifices the circuit's fault coverage, uses a ***pseudo-random pattern generator*** (PRPG) for generating a pseudo-random sequence of test patterns [Bardell 1987; Rajski 1998; Bushnell 2000; Jha 2003]. **Pseudo-random testing** has the advantage of being applicable to both sequential and combinational circuits; however, there are difficulties in determining the required test length and fault coverage.

**Table 3.5** Primitive Polynomials of Degree *n* up to 100

| *n* | Exponents | *n* | Exponents | *n* | Exponents | *n* | Exponents |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 26 | 8 7 1 0 | 51 | 16 15 1 0 | 76 | 36 35 1 0 |
| 2 | 1 0 | 27 | 8 7 1 0 | 52 | 3 0 | 77 | 31 30 1 0 |
| 3 | 1 0 | 28 | 3 0 | 53 | 16 15 1 0 | 78 | 20 19 1 0 |
| 4 | 1 0 | 29 | 2 0 | 54 | 37 36 1 0 | 79 | 9 0 |
| 5 | 2 0 | 30 | 16 15 1 0 | 55 | 24 0 | 80 | 38 37 1 0 |
| 6 | 1 0 | 31 | 3 0 | 56 | 22 21 1 0 | 81 | 4 0 |
| 7 | 1 0 | 32 | 28 27 1 0 | 57 | 7 0 | 82 | 38 35 3 0 |
| 8 | 6 5 1 0 | 33 | 13 0 | 58 | 19 0 | 83 | 46 45 1 0 |
| 9 | 4 0 | 34 | 15 14 1 0 | 59 | 22 21 1 0 | 84 | 13 0 |
| 10 | 3 0 | 35 | 2 0 | 60 | 1 0 | 85 | 28 27 1 0 |
| 11 | 2 0 | 36 | 11 0 | 61 | 16 15 1 0 | 86 | 13 12 1 0 |
| 12 | 7 4 3 0 | 37 | 12 10 2 0 | 62 | 57 56 1 0 | 87 | 13 0 |
| 13 | 4 3 1 0 | 38 | 6 5 1 0 | 63 | 1 0 | 88 | 72 71 1 0 |
| 14 | 12 11 1 0 | 39 | 4 0 | 64 | 4 3 1 0 | 89 | 38 0 |
| 15 | 1 0 | 40 | 21 19 2 0 | 65 | 18 0 | 90 | 19 18 1 0 |
| 16 | 5 3 2 0 | 41 | 3 0 | 66 | 10 9 1 0 | 91 | 84 83 1 0 |
| 17 | 3 0 | 42 | 23 22 1 0 | 67 | 10 9 1 0 | 92 | 13 12 1 0 |
| 18 | 7 0 | 43 | 6 5 1 0 | 68 | 9 0 | 93 | 2 0 |
| 19 | 6 5 1 0 | 44 | 27 26 1 0 | 69 | 29 27 2 0 | 94 | 21 0 |
| 20 | 3 0 | 45 | 4 3 1 0 | 70 | 16 15 1 0 | 95 | 11 0 |
| 21 | 2 0 | 46 | 21 20 1 0 | 71 | 6 0 | 96 | 49 47 2 0 |
| 22 | 1 0 | 47 | 5 0 | 72 | 53 47 6 0 | 97 | 6 0 |
| 23 | 5 0 | 48 | 28 27 1 0 | 73 | 25 0 | 98 | 11 0 |
| 24 | 4 3 1 0 | 49 | 9 0 | 74 | 16 15 1 0 | 99 | 47 45 2 0 |
| 25 | 3 0 | 50 | 27 26 1 0 | 75 | 11 10 1 0 | 100 | 37 0 |

*Note: "24 4 3 1 0" means* $p(x) = x^{24} + x^4 + x^3 + x^1 + x^0 = x^{24} + x^4 + x^3 + x + 1$.

#### 3.4.1.2.1 Maximum-length LFSR

Maximum-length LFSRs are commonly used for pseudo-random pattern generation. Each LFSR produces a sequence with 0.5 probability of generating 1's

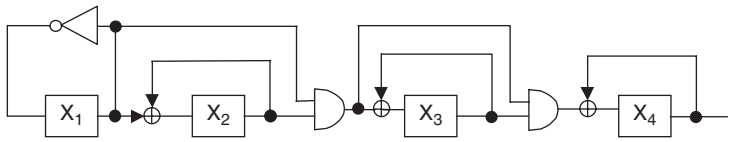



**FIGURE 3.17**

Example binary counter as EPG.

(or with probability distribution 0.5) at every output. The **LFSR pattern generation technique** that uses these LFSRs, in standard or modular form, to generate patterns for the entire design has the advantage of being very easy to implement. The major problem with this approach is that some circuits may be ***random pattern resistant*** (RP-resistant). For instance, consider a 5-input OR gate. The probability of applying an all-zero pattern to all inputs is 1/32. This makes it difficult to test the RP-resistant OR-gate output stuck-at-1.

#### 3.4.1.2.2 Weighted LFSR

It is possible to increase fault coverage (and detect most RP-resistant faults) in RP-resistant designs. A **weighted pattern generation technique** that uses an LFSR and a combinational circuit was first described in [Schnurmann 1975]. The combinational circuit inserted between the output of the LFSR and the CUT is to increase the frequency of occurrence of one logic value while decreasing the other logic value. This approach may increase the probability of detecting those faults that are hard to detect with the typical LFSR pattern generation technique.

Implementation methods for realizing this scheme are further discussed in [Chin 1984]. The weighted pattern generation technique described in that paper modifies the maximum-length LFSR to produce an equally weighted distribution of 0's and 1's at the input of the CUT. It skews the LFSR probability distribution of 0.5 to either 0.25 or 0.75 to increase the chance of detecting those faults that are hard to detect with just a 0.5 distribution. Better fault coverage was also found in [Wunderlich 1987], where probability distributions in a multiple of 0.125 (rather than 0.25) are used. Figure 3.18 shows a four-stage weighted (maximum-length) LFSR with probability distribution 0.25 [Chin 1984].

#### 3.4.1.2.3 Cellular automata

**Cellular automata** were first introduced in [Wolfram 1983]. They yielded better randomness property than LFSRs [Hortensius 1989]. The *cellular automaton based* (or CA-based) *pseudo-random pattern generator* (PRPG) is attractive for BIST applications [Khara 1987; Gloster 1988; Wang 1989; van Sas 1990] because it (1) provides patterns that look *more* random at the circuit inputs, (2) has higher opportunity to reach very high fault coverage in a circuit that is RP-resistant, and

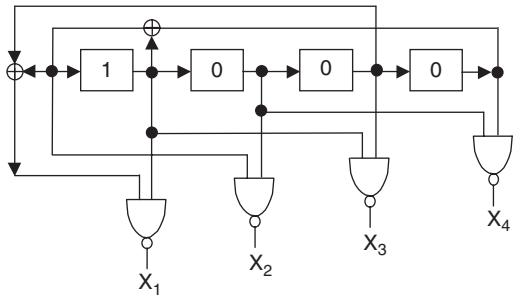



**FIGURE 3.18**

Example weighted LFSR as PRPG.

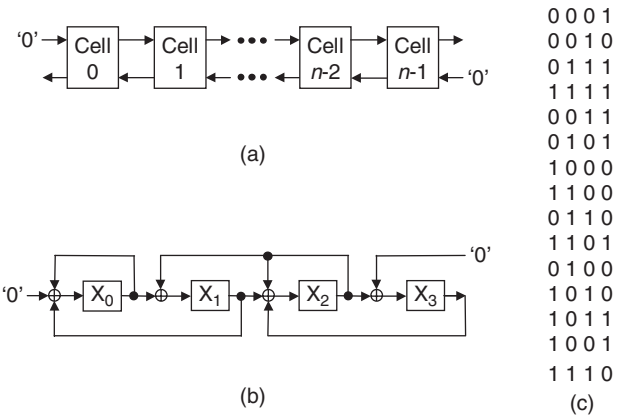



**FIGURE 3.19**

Example cellular automaton (CA) as PRPG: (a) General structure of an $n$-stage CA. (b) Four-stage CA. (c) Test sequence generated by (b).

(3) has implementation advantages because it only requires adjacent neighbor communication (no global feedback unlike the modular LFSR case).

A *cellular automaton* (CA) is a collection of **cells** with forward and backward connections. A general structure is shown in Figure 3.19a. Each cell can only connect to its local neighbors (adjacent left and right cells). The connections are expressed as **rules**; each rule determines the next state of a cell on the basis of the state of the cell and its neighbors. Assume cell $i$ can only talk with its neighbors, $i-1$ and $i+1$. Define:

$$\textit{Rule 90}: x_i(t+1) = x_{i-1}(t) + x_{i+1}(t)$$

and

$$\textit{Rule 150}: x_i(t+1) = x_{i-1}(t) + x_i(t) + x_{i+1}(t)$$

Then the two rules, *rule 90* and *rule 150*, can be established on the basis of the following state transition table:

| $x_{i-1}(t)x_i(t)x_{i+1}(t)$ | 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
|---|---|---|---|---|---|---|---|---|
| *Rule 90:* $x_i(t+1)$ | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| | $2^6$ | + $2^4$ | + $2^3$ | + | $2^1 = 90$ | | | |
| *Rule 150:* $x_i(t+1)$ | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| | $2^7$ | + $2^4$ | + $2^2$ | + | $2^1 = 150$ | | | |

The terms *rule 90* and *rule 150* were derived from their decimal equivalents of the binary code for the next state of cell $i$ [Hortensius 1989]. Figure 3.19b shows an example of a four-stage CA generated by alternating rules 150 (on even cells) and 90 (on odd cells). Similar to the four-stage modular LFSR given in Figure 3.16b, the four-stage CA generates a *maximum-length sequence* of 15 distinct states as listed in Figure 3.19c.

It has been shown in [Hortensius 1989] that by combining cellular automata rules 90 and 150, an $n$-stage CA can generate a maximum-length sequence of $2^n-1$. The construction rules for $4 \leq n \leq 53$ can be found in [Hortensius 1989] and are listed in Table 3.6.

The CA-based PRPG can be programmed as a **universal CA** for generating different orders of test sequences. A **universal CA-cell** for generating patterns on the basis of *rule 90* or *rule 150* is given in Figure 3.20 [Wang 1989]. When the RULE150_SELECT signal is set to 1, the universal CA-cell will behave as a *rule 150* cell; otherwise, it will act as a *rule 90* cell. This *universal CA* structure is useful for BIST applications where it is required to obtain very high fault coverage for RP-resistant designs or detect additional classes of faults.

### 3.4.1.3 *Pseudo-exhaustive testing*

Another approach to reduce the test time to a practical value while retaining many of the advantages of exhaustive testing is the **pseudo-exhaustive test technique**. It applies fewer than $2^n$ test patterns to an $n$-input combinational CUT. The technique depends on whether any output is driven by all of its inputs. If none of the outputs depends on all inputs, a **verification test approach** proposed in [McCluskey 1984] can be used to test these circuits. In circuits in which there is one output that depends on all inputs or the test time that uses verification testing is still too long, a **segmentation test approach** must be used [McCluskey 1981]. Pseudo-exhaustive testing guarantees single-stuck fault coverage without any detailed circuit analysis.

**Table 3.6** Construction Rules for Cellular Automat of Length *n* up to 53

| *n* | Rule* | *n* | Rule* |
|---|---|---|---|
| 4 | 05 | 29 | 2,512,712103 |
| 5 | 31 | 30 | 7,211,545,075 |
| 6 | 25 | 31 | 04,625,575,630 |
| 7 | 152 | 32 | 10,602,335,725 |
| 8 | 325 | 33 | 03,047,162,605 |
| 9 | 625 | 34 | 036,055,030,672 |
| 10 | 0,525 | 35 | 127,573,165,123 |
| 11 | 3,252 | 36 | 514,443,726,043 |
| 12 | 2,252 | 37 | 0,226,365,530,263 |
| 13 | 14,524 | 38 | 0,345,366,317,023 |
| 14 | 17,576 | 39 | 6,427,667,463,554 |
| 15 | 44,241 | 40 | 00,731,257,441,345 |
| 16 | 152,525 | 41 | 15,376,413,143,607 |
| 17 | 175,763 | 42 | 11,766,345,114,746 |
| 18 | 252,525 | 43 | 035,342,704,132,622 |
| 19 | 0,646,611 | 44 | 074,756,556,045,302 |
| 20 | 3,635,577 | 45 | 151,315,510,461,515 |
| 21 | 3,630,173 | 46 | 0,112,312,150,547,326 |
| 22 | 05,252,525 | 47 | 0,713,747,124,427,015 |
| 23 | 32,716,532 | 48 | 0,606,762,247,217,017 |
| 24 | 77,226,526 | 49 | 02,675,443,137,056,631 |
| 25 | 136,524,744 | 50 | 23,233,006,150,544,226 |
| 26 | 132,642,730 | 51 | 04,135,241,323,505,027 |
| 27 | 037,014,415 | 52 | 031,067,567,742,172,706 |
| 28 | 0,525,252,525 | 53 | 207,121,011,145,676,625 |

**Rule is given in octal format. For n = 7,* Rule = 152 = 001,101,010 = 1,101,010, where "0" denotes a *rule 90* cell and "1" denotes a *rule 150* cell, or vice versa.

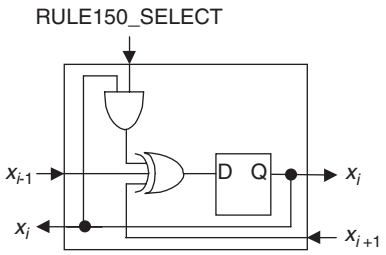



**FIGURE 3.20**

A universal CA-cell structure.

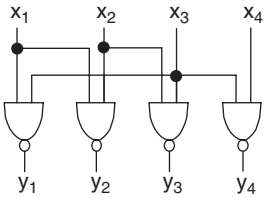



**FIGURE 3.21**

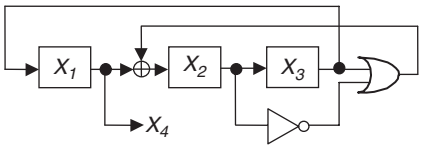
An $(n,w) = (4,2)$ CUT.

Verification testing [McCluskey 1984] divides the circuit under test into $m$ cones, where $m$ is the number of outputs. It is based on backtracing from each circuit output to determine the actual number of inputs that drive the output. Each cone will receive exhaustive test patterns, and all cones are tested concurrently.

Assume the combinational CUT has $n$ inputs and $m$ outputs. Let $w$ be the maximum number of input variables on which any output of the CUT depends. Then, the $n$-input $m$-output combinational CUT is defined as an $(n,w)$ CUT, where $w < n$. Figure 3.21 shows an $(n,w) = (4,2)$ CUT that will be used as an example for designing the *pseudo-exhaustive pattern generators* (PEPGs).

#### 3.4.1.3.1 Syndrome driver counter

The first method for **pseudo-exhaustive pattern generation** was proposed in [Savir 1980]. ***Syndrome driver counters*** (SDCs) are used to generate test patterns [Barzilai 1981]. The SDC can be a binary counter, a maximum-length LFSR, or a complete LFSR. This method checks whether some circuit inputs can share the same test signal. If $n$-$p$ inputs, $p < n$, can share the **test signals** with the other $p$ inputs, then the circuit can be tested exhaustively with these $p$ inputs. In this case, the test length becomes $2^p$ if $p = w$, or $2^p - 1$ if $p > w$. Figure 3.22 shows a three-stage SDC used to test the circuit given in Figure 3.21. Because both inputs $x_1$ and $x_4$ do




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

**FIGURE 3.22**

Example syndrome driver counter as PEPG.

not drive the same output, one test signal can be used to drive both inputs. In this case, $p$ is 3, and the test length becomes $2^3 - 1 = 7$. Designs based on the SDC method for in-circuit test pattern generation are simple. The problem with this method is that when $p$ is close to $n$, it may still take too long to test the circuit.

### 3.4.1.3.2 Condensed LFSR

The problem can be solved by use of the **condensed LFSR** approach proposed in [Wang 1986a]. Condensed LFSRs are constructed on the basis of **linear codes** [Peterson 1972]. An $(n,k)$ *linear code over GF*(2) generates a code space $C$ containing $2^k$ distinct code words ($n$-tuples) with the following property: if $c_1 \in C$ and $c_2 \in C$, then $c_1 + c_2 \in C$. Define an $(n,k)$ *condensed LFSR* as an $n$-stage modular LFSR with period $2^k - 1$. A condensed LFSR for testing an $(n,w)$ CUT is constructed by first computing the smallest integer $k$ such that:

$$w \leq \lceil k/(n-k+1) \rceil + \lfloor k/(n-k+1) \rfloor$$

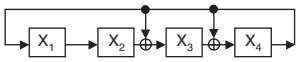
where $\lceil x \rceil$ denotes the smallest integer equal to or greater than the real number $x$, and $\lfloor y \rfloor$ denotes the largest integer equal to or smaller than the real number $y$.

Then, by use of:

$$f(x) = g(x)p(x) = (1 + x + x^2 + \ldots + x^{n-k})p(x)$$

an $(n,k)$ *condensed LFSR* can be realized, where $g(x)$ is a **generator polynomial** of degree $n$-$k$ generating the $(n,k)$ linear code, and $p(x)$ is a primitive polynomial of degree $k$.

Consider the $(n,k) = (4,3)$ condensed LFSR shown in Figure 3.23a used to test the $(n,w) = (4,2)$ CUT. Because $n = 4$ and $w = 2$, we obtain $k = 3$ and




(a)

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

(b)

**FIGURE 3.23**

Example condensed LFSR as PEPG: (a) (4,3) condensed LFSR. (b) Test sequence generated by (a).

$(n - k) = 1$. Selecting $p(x) = 1 + x + x^3$, we have $f(x) = (1 + x)(1 + x + x^3) = 1 + x^2 + x^3 + x^4$. Figure 3.23b lists the generated period-7 test sequence. It is important to note that the seed polynomial $S_0(x)$ of the LFSR must be divisible by $g(x)$. In the example, we set $S_0(x) = g(x) = 1 + x$, or $S_0$ to {1100}.

For any given $(n,w)$ CUT, this method uses at most two seeds and has shown to be effective when $w \geq n/2$. Designs based on this method are simple. However, this technique uses more patterns than the **combined LFSR/SR** approach, which uses a combination of an LFSR and a *shift register* (SR) [Barzilai 1983; Tang 1984; Chen 1987] and the **cyclic LFSR** approach [Wang 1987, 1988b] when $w < n/2$. For other verification test approaches, refer to [Abramovici 1994; Wang 2006a].

## 3.4.2 Output response analysis

For scan designs, our assumption was that output responses coming out of the *circuit under test* (CUT) are compared directly on a tester. For BIST operations, it is impossible to store all output responses on-chip, on-board, or in-system to perform bit-by-bit comparison. An *output response analysis* technique must be used such that output responses can be compacted into a **signature** and compared with a *golden signature* for the fault-free circuit either embedded on-chip or stored off-chip.

*Compaction* differs from *compression* in that compression is loss-less, whereas compaction is lossy. **Compaction** is a method for dramatically reducing the number of bits in the original circuit response during testing in which some information is lost. **Compression** is a method for reducing the number of bits in the original circuit response in which no information is lost, such that the original output sequence can be fully regenerated from the compressed sequence [Bushnell 2000]. Because all output response analysis schemes involve information loss, they are referred to as *output response compaction*. However, there is no general consensus in academia yet as to when the terms compaction or compression are to be used. However, for output response analysis, throughout the book, we will refer to the lossy compression as compaction.

In this section, we will present three different output response compaction techniques: (1) **ones count testing**, (2) **transition count testing**, and (3) **signature analysis**. We will also describe the architectures of the *output response analyzers* (ORAs) that are used. The signature analysis technique will be described in more detail, because it is the most popular compaction technique in use today.

When compaction is used, it is important to ensure that the faulty and fault-free signatures are different. If they are the same, the fault(s) can go undetected. This situation is referred to as **error masking**, and the erroneous output response is said to be an **alias** of the correct output response [Abramovici 1994]. It is also important to ensure that none of the output responses contains an unknown ($X$) value. If an unknown value is generated and propagated directly or indirectly to the ORA, then the ORA can no longer function reliably. Therefore, it is required that all unknown ($X$) propagation problems be fixed to ensure that the logic BIST system will operate correctly. Such **X-blocking** or **X-bounding** techniques have been extensively discussed in [Wang 2006a].

### 3.4.2.1 *Ones count testing*

Assume that the CUT has only one output and the output contains a stream of $L$ bits. Let the fault-free output response, $R_0$, be $\{r_0\, r_1\, r_2 \ldots r_{L-1}\}$. The **ones count test technique** will only need a counter to count the number of 1's in the bit stream. For instance, if $R_0 = \{0101100\}$, then the signature or ones count of $R_0$, $OC(R_0)$, is 3. If fault $f_1$ present in the CUT causes an erroneous response $R_1 = \{1100110\}$, then it will be detected because $OC(R_1) = 4$. However, fault $f_2$ causing $R_2 = \{0101010\}$ will not be detected because $OC(R_2) = OC(R_0) = 3$. Let the fault-free signature or ones count be $m$. There will be $C(L,m)$ possible ways having $m$ 1's in an $L$-bit stream. Assuming all faulty sequences are equally likely to occur as the response of the CUT, the **aliasing probability** or **masking probability** of the use of ones count testing having $m$ 1's [Savir 1985] can be expressed as

$$P_{OC}(m) = \Big(C(L,m) - 1\Big)/(2^L - 1)$$

In the previous example, where $m = OC(R_0) = 3$ and $L = 7$, $P_{OC}(m) = 34/127 = 0.27$. Figure 3.24 shows the ones count test circuit for testing the CUT with $T$ patterns. The number of stages in the counter design must be equal to or greater than $\lceil \log_2(L+1) \rceil$.

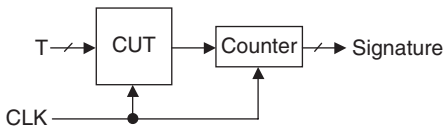



**FIGURE 3.24**

Ones counter as ORA.

### 3.4.2.2 *Transition count testing*

The theory behind transition count testing is similar to that for ones count testing, except the signature is defined as the number of 0-to-1 and 1-to-0 transitions. The **transition count test technique** [Hayes 1976] simply requires the use of a D flip-flop and an XOR gate connected to a ones counter (see Figure 3.25) to count the number of transitions in the output data stream. Consider the example given previously. Because $R_0 = \{0101100\}$, the signature or transition count of $R_0$, $TC(R_0)$, will be 4. Assume that the initial state of the D flip-flop, $r_{-1}$, is 0. Fault $f_1$ causing an erroneous response $R_1 = \{1100110\}$ will not be detected because $TC(R_1) = TC(R_0) = 4$, whereas fault $f_2$ causing $R_2 = \{0101010\}$ will be detected because $TC(R_2) = 6$.

Let the fault-free signature or transition count be $m$. Because a given $L$-bit sequence $R_0$ that starts with $r_0 = 0$ has $L - 1$ possible transitions, the number of sequences with $m$ transitions can be given by $C(L - 1,m)$. Because $R_0$ can also start with $r_0 = 1$, there will be a total of $2C(L - 1,m)$ possible ways having $m$ 0-to-1 and 1-to-0 transitions in an $L$-bit stream. Assuming all faulty sequences are equally likely to occur as the response of the CUT, the *aliasing probability* or *masking probability* of the use of transition count testing having $m$ transitions [Savir 1985] is

$$P_{TC}(m) = \Big(2C(L-1,m) - 1\Big)/(2^L - 1)$$

In the previous example, where $m = TC(R_0) = 4$ and $L = 7$, $P_{TC}(m) = 29/127 = 0.23$. Figure 3.25 shows the transition count test circuit. The number of stages in the counter design must be equal to or greater than $\lceil \log_2(L+1) \rceil$.

### 3.4.2.3 *Signature analysis*

Signature analysis is the most popular response compaction technique used today. The compaction scheme, based on **cyclic redundancy checking** (CRC) [Peterson 1972], was first developed in [Benowitz 1975]. Hewlett-Packard commercialized the first logic analyzer, called HP 5004A Signature Analyzer, based on the scheme and referred to it as **signature analysis** [Frohwerk 1977].

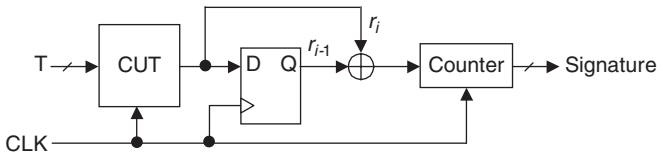



**FIGURE 3.25**

Transition counter as ORA.

In this subsection, we will discuss two signature analysis schemes: (1) **serial signature analysis** for compacting responses from a CUT having a single output and (2) **parallel signature analysis** for compacting responses from a CUT having multiple outputs.

### 3.4.2.3.1 Serial Signature Analysis

Consider the *n*-stage ***single-input signature register*** (SISR) shown in Figure 3.26. This SISR uses an additional XOR gate at the input for compacting an *L*-bit output sequence, *M*, into the *modular LFSR*. Let $M = \{m_0\, m_1\, m_2 \ldots m_{L-1}\}$, and define:

$$M(x) = m_0 + m_1 x + m_2 x^2 + \ldots + m_{L-1} x^{L-1}$$

After shifting the *L*-bit output sequence, *M*, into the *modular LFSR*, the contents (remainder) of the SISR, *R*, is given as $\{r_0\, r_1\, r_2 \ldots r_{n-1}\}$, or

$$r(x) = r_0 + r_1 x + r_2 x^2 + \ldots + r_{n-1} x^{n-1}$$

The SISR is basically a *CRC code generator* [Peterson 1972] or a *cyclic code checker* [Benowitz 1975]. Let the *characteristic polynomial* of the modular LFSR be $f(x)$. The authors in [Peterson 1972] have shown that the SISR performs polynomial division of $M(x)$ by $f(x)$, or

$$M(x) = q(x)f(x) + r(x)$$

The final state or **signature** in the SISR is the *polynomial remainder, r(x)*, of the division. Consider the four-stage SISR given in Figure 3.27 with $f(x) = 1 + x + x^4$. Assuming $M = \{10011011\}$, we can express $M(x) = 1 + x^3 + x^4 + x^6 + x^7$. By use of polynomial division, we obtain $q(x) = x^2 + x^3$ and $r(x) = 1 + x^2 + x^3$ or $R = \{1011\}$. The remainder {1011} is equal to the *signature* derived from Figure 3.27a when the SISR is first initialized to a *starting pattern* (*seed*) of {0000}.

Now, assume fault $f_1$ produces an erroneous output stream $M' = \{11001011\}$ or $M'(x) = 1 + x + x^4 + x^6 + x^7$, as given in Figure 3.27b. By use of polynomial division, we obtain $q'(x) = x^2 + x^3$ and $r'(x) = 1 + x + x^2$ or $R' = \{1110\}$. Because the faulty signature *R′*, {1110}, is different from the fault-free signature *R*, {1011}, fault $f_1$ is detected. For fault $f_2$ with $M'' = \{11001101\}$ or $M''(x) = 1 + x + x^4 + x^5 + x^7$ as given in Figure 3.27c, we have $q''(x) = x + x^3$ and $r''(x) = 1 + x^2 + x^3$ or $R'' = \{1011\}$. Because $R'' = R$, fault $f_2$ is not detected.

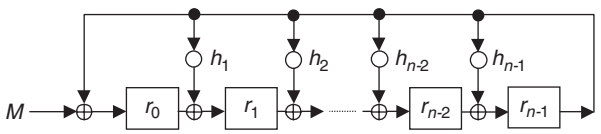
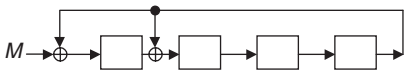



**FIGURE 3.26**

An *n*-stage single-input signature register (SISR).




| $M$ | $r_0$ | $r_1$ | $r_2$ | $r_3$ |
|---|---|---|---|---|
| 1 | **0** | **0** | **0** | **0** |
| 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| $R$ | **1** | **0** | **1** | **1** |

(a)

| $M'$ | $r_0$ | $r_1$ | $r_2$ | $r_3$ |
|---|---|---|---|---|
| 1 | **0** | **0** | **0** | **0** |
| 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| *0* | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| *1* | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| $R'$ | **1** | **1** | **1** | **0** |

(b)

| $M''$ | $r_0$ | $r_1$ | $r_2$ | $r_3$ |
|---|---|---|---|---|
| 1 | **0** | **0** | **0** | **0** |
| *0* | 1 | 0 | 0 | 0 |
| *1* | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| *0* | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| *1* | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| $R''$ | **1** | **0** | **1** | **1** |

(c)

**FIGURE 3.27**

A four-stage SISR: (a) Fault-free signature. (b) Signature for fault $f_1$. (c) Signature for fault $f_2$.

The *fault detection* or *aliasing* problem of an SISR can be better understood by looking at the *error sequence E* or *error polynomial E(x)* of the fault-free sequence $M$ and a faulty sequence $M'$. Define $E = M + M'$, or:

$$E(x) = M(x) + M'(x)$$

If $E(x)$ is not divisible by $f(x)$, then all faults generating the faulty sequence $M'$ will be detected. Otherwise, these faults are not detected. Consider fault $f_1$ again. We obtain $E = \{01010000\} = M + M' = \{10011011\} + \{11001011\}$ or $E(x) = x + x^3$. Because $E(x)$ is not divisible by $f(x) = 1 + x + x^4$, fault $f_1$ is detected. Consider fault $f_2$ again. We have $E = \{01010110\} = M + M'' = \{10011011\} + \{11001101\}$ or $E(x) = x + x^3 + x^5 + x^6$. Because $f(x)$ divides $E(x)$, *i.e.,* $E(x) = (x + x^2)f(x)$, fault $f_2$ is not detected.

Assume the SISR consists of $n$ stages. For a given $L$-bit sequence, $L > n$, there are $2^{(L-n)}$ possible ways of producing an $n$-bit signature of which one is the correct signature. Because there are a total of $2^L - 1$ erroneous sequences in an $L$-bit stream, the *aliasing probability* with an $n$-stage SISR for *serial signature analysis* (SSA) is:

$$P_{SSA}(n) = \left(2^{(L-n)} - 1\right)/(2^L - 1)$$

If $L >> n$, then $P_{SSA}(n) \approx 2^{-n}$. When $n = 20$, $P_{SSA}(n) < 2^{-20} = 0.0001\%$.

#### 3.4.2.3.2 Parallel Signature Analysis

A common problem when using ones count testing, transition count testing, and serial signature analysis is the excessive hardware cost required to test an $m$-output CUT. It is possible to reduce the hardware cost by use of an $m$-to-1 multiplexer, but this increases the test time $m$ times.

Consider the *n*-stage ***multiple-input signature register*** (MISR) shown in Figure 3.28. The MISR uses *n* extra XOR gates for compacting *n L*-bit output sequences, $M_0$ *to* $M_{n-1}$, into the *modular LFSR* simultaneously.

[Hassan 1984] has shown that the *n*-input MISR can be remodeled as a single-input SISR with *effective input sequence* $M(x)$ and *effective error polynomial* $E(x)$ expressed as:

$$M(x) = M_0(x) + xM_1(x) + \ldots + x^{n-2}M_{n-2}(x) + x^{n-1}M_{n-1}(x)$$

and

$$E(x) = E_0(x) + xE_1(x) + \ldots + x^{n-2}E_{n-2}(x) + x^{n-1}E_{n-1}(x)$$

Consider the four-stage MISR shown in Figure 3.29 that uses $f(x) = 1 + x + x^4$. Let $M_0 = \{10010\}$, $M_1 = \{01010\}$, $M_2 = \{11000\}$, and $M_3 = \{10011\}$. From this information, the signature $R$ of the MISR can be calculated as {1011}. With $M(x) = M_0(x) + xM_1(x) + x^2M_2(x) + x^3M_3(x)$, we obtain $M(x) = 1 + x^3 + x^4 + x^6 + x^7$ or $M = \{10011011\}$ as shown in Figure 3.30. This is the same data stream we used in the SISR example in Figure 3.27a. Therefore, $R = \{1011\}$.

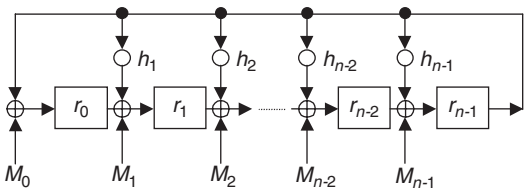



**FIGURE 3.28**

An *n*-stage multiple-input signature register (MISR).

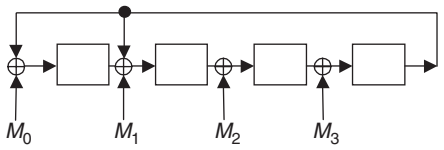



**FIGURE 3.29**

A four-stage MISR.

| | |
|---|---|
| $M_0$ | 1 0 0 1 0 |
| $M_1$ | 0 1 0 1 0 |
| $M_2$ | 1 1 0 0 0 |
| $M_3$ | 1 0 0 1 1 |
| $M$ | 1 0 0 1 1 0 1 1 |

**FIGURE 3.30**

An equivalent *M* sequence.

Assume there are $m$ $L$-bit sequences to be compacted in an $n$-stage MISR, where $L > n \geq m \geq 2$. The *aliasing probability* for *parallel signature analysis* (PSA) now becomes:

$$P_{PSA}(n) = \left(2^{(mL-n)} - 1\right)/(2^{mL} - 1)$$

If $L >> n$, *then* $P_{PSA}(n) \approx 2^{-n}$. When $n = 20$, $P_{PSA}(n) < 2^{-20} = 0.0001\%$. The result suggests that $P_{PSA}(n)$ mainly depends on $n$, when $L >> n$. Hence, increasing the number of MISR stages or the use of the same MISR but with a different $f(x)$ can substantially reduce the *aliasing probability* [Hassan 1984; Williams 1987].

## 3.4.3 Logic BIST architectures

Several architectures for incorporating **offline BIST** techniques into a design have been proposed. These BIST architectures can be classified into two classes: (1) those that use the **test-per-scan BIST** scheme and (2) those that use the **test-per-clock BIST** scheme. The *test-per-scan BIST* scheme takes advantage of the already built-in scan chains of the scan design and applies a test pattern to the CUT after a shift operation is completed; hence, the hardware overhead is low. The *test-per-clock BIST* scheme, however, applies a test pattern to the CUT and captures its test response every system clock cycle; hence, the scheme can execute tests much faster than the test-per-scan BIST scheme but at an expense of more hardware overhead.

In this subsection, we only discuss three representative BIST architectures, the first two for pseudo-random testing and the last for pseudo-exhaustive testing. Although pseudo-random testing is commonly adopted in industry, the exhaustive and pseudo-exhaustive test techniques are applicable for designs that use the test-per-clock BIST scheme. For a more comprehensive survey of these BIST architectures, refer to [Abramovici 1994; Bardell 1987; McCluskey 1985; Wang 2006a]. Fault coverage enhancement with the pseudo-random test technique can also be found in [Tsai 1999; Wang 2006a; Lai 2007].

### 3.4.3.1 *Self-testing with MISR and parallel SRSG (STUMPS)*

A test-per-scan BIST design was presented in [Bardell 1982]. This design, shown in Figure 3.31, contains a PRPG (parallel ***shift register sequence generator*** [SRSG]) and a MISR. The scan chains are loaded in parallel from the PRPG. The system clocks are then triggered, and the test responses are shifted to the MISR for compaction. New test patterns are shifted in at the same time while test responses are being shifted out. This BIST architecture that uses the test-per-scan BIST scheme is referred to as ***self-testing with MISR and parallel SRSG*** **(STUMPS)** [Bardell 1982].

Because of the ease of integration with traditional scan architecture, the **STUMPS** architecture is the only BIST architecture widely used in industry to

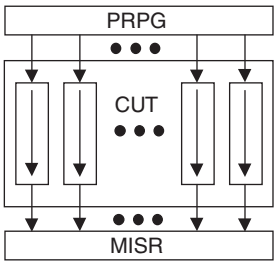



**FIGURE 3.31**

STUMPS.

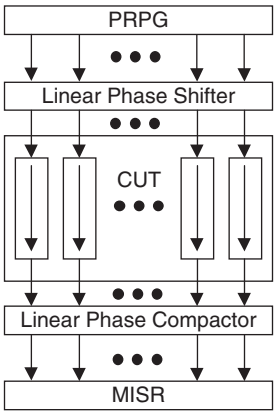



**FIGURE 3.32**

A STUMPS-based architecture.

date. To further reduce the lengths of the PRPG and MISR and improve the randomness of the PRPG, a STUMPS-based architecture that includes an optional linear phase shifter and an optional linear phase compactor is often used in industrial applications [Nadeau-Dostie 2000; Cheon 2005]. The linear phase shifter and linear phase compactor typically comprise a network of XOR gates. Figure 3.32 shows the STUMPS-based architecture.

### 3.4.3.2 *Built-in logic block observer (BILBO)*

The architecture described in [Könemann 1979, 1980] applies to circuits that can be partitioned into independent modules (logic blocks). Each module is assumed to have its own input and output registers (storage elements), or such registers are added to the circuit where necessary. The registers are redesigned so that for test purposes they act as PRPGs for test generation or MISRs for signature analysis. The redesigned register is called a ***built-in logic block observer*** (BILBO).

The BILBO is operated in four modes: normal mode, scan mode, test generation or signature analysis mode, and reset mode. A typical three-stage BILBO, which is reconfigurable into a TPG or a MISR during self-test is shown in Figure 3.33. It is controlled by two control inputs $B_1$ and $B_2$. When both control inputs $B_1$ and $B_2$ are equal to 1, the circuit functions in normal mode with the inputs $Y_i$ gated directly into the D flip-flops. When both control inputs are equal to 0, the BILBO is configured as a shift register. Test data can be shifted in through the serial scan-in port or shifted out through the serial scan-out port. Setting $B_1 = 1$ and $B_2 = 0$ converts the BILBO into a MISR. It can then be used in this configuration as a TPG by holding every $Y_i$ input to 1. The BILBO is reset after a system clock is triggered when $B_1 = 0$ and $B_2 = 1$.

This technique is most suitable for testing circuits, such as *random-access memories* (RAMs), *read-only memories* (ROMs), or bus-oriented circuits, where input and output registers of the partitioned modules can be reconfigured independently. For testing finite-state machines or pipeline-oriented circuits as shown in Figure 3.34, the signature data from the previous module must be

| $B_1$ | $B_2$ | Operation mode |
|---|---|---|
| 1 | 1 | Normal |
| 0 | 0 | Scan |
| 1 | 0 | Mixed Test Generation and Signature Analysis |
| 0 | 1 | Reset |

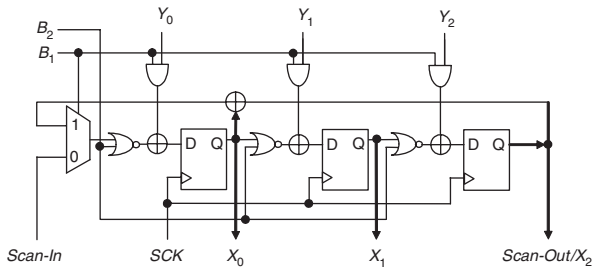



**FIGURE 3.33**

A three-stage built-in logic block observer (BILBO).

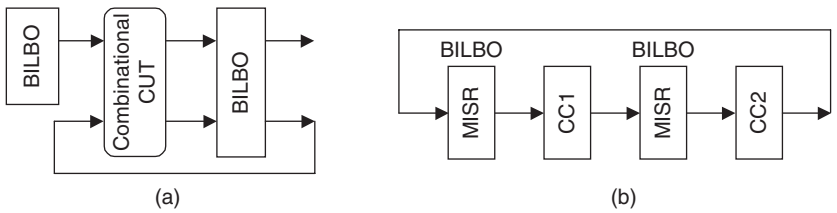



**FIGURE 3.34**

BILBO architectures: (a) For testing a finite-state machine. (b) For testing a pipeline-oriented circuit.

used as test patterns for the next module, because the test generation and signature analysis modes cannot be separated. In this case, a detailed fault simulation is required to achieve 100% single-stuck fault coverage.

#### 3.4.3.3 *Concurrent built-in logic block observer (CBILBO)*

One technique to overcome the above BILBO fault coverage loss problem is to use the ***concurrent built-in logic block observer*** (CBILBO) approach [Wang 1986b]. Reconfigured from the BILBO design, the CBILBO is based on the test-per-clock BIST scheme and uses two registers to perform test generation and signature analysis simultaneously. A CBILBO design is illustrated in Figure 3.35, where only three modes of operation are considered: normal, scan, and test generation and signature analysis. When $B_1 = 0$ and $B_2 = 1$, the upper D flip-flops act as a MISR for signature analysis, whereas the lower two-port D flip-flops form a TPG for test generation. Because signature analysis is separated from test generation, an *exhaustive* or *pseudo-exhaustive pattern generator* (EPG/PEPG) can now be used for test generation; therefore, no fault simulation is required, and it is possible to achieve 100% single-stuck fault coverage with the CBILBO architectures for testing designs shown in Figure 3.36. However, the hardware cost associated with the use of the CBILBO approach is generally higher than for the STUMPS approach.

### 3.4.4 Industry practices

Logic BIST has a history of more than 30 years since its invention in the 1970s. Although it is only a few years behind the invention of scan, logic BIST has yet

| $B_1$ | $B_2$ | **Operation mode** |
|---|---|---|
| | 0 | Normal |
| 1 | 1 | Scan |
| 0 | 1 | Test Generation and Signature Analysis |

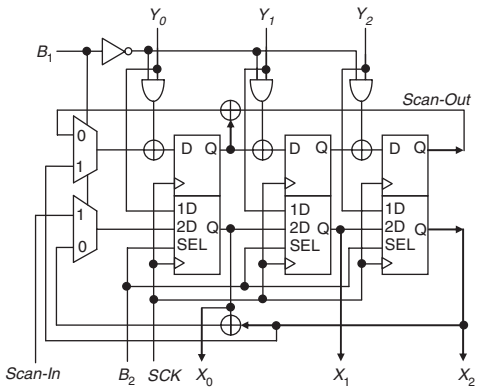



**FIGURE 3.35**

A three-stage concurrent BILBO (CBILBO).

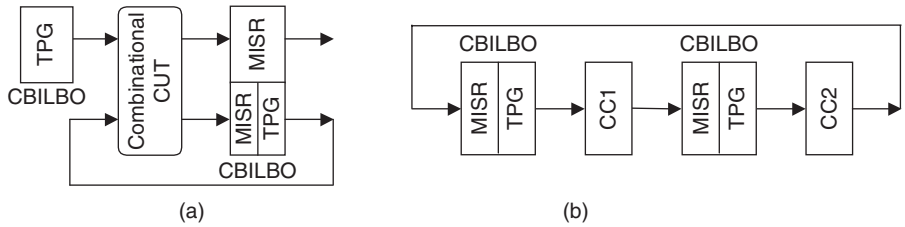



**FIGURE 3.36**

CBILBO architectures: (a) For testing a finite-state machine. (b) For testing a pipeline-oriented circuit.

to gain strong industry support. The worldwide market is estimated to be close to 10% of the scan market. The logic BIST products available in the marketplace now include **Encounter Test** from Cadence Design Systems [Cadence 2008], **ETLogic** from LogicVision [LogicVision 2008], **LBIST Architect** from Mentor Graphics [Mentor 2008], and **TurboBIST-Logic** from SynTest Technologies [SynTest 2008]. The logic BIST product offered in Encounter Test by Cadence currently includes support for test structure extraction, verification, logic simulation for signatures, and fault simulation for coverage. Unlike all other three BIST vendors that provide their own logic BIST structures in their respective products, Cadence offers a service to insert custom logic BIST structures or to use any customer-inserted logic BIST structures, including working with the customer to have custom on-chip clocking for logic BIST. A similar case exists in ETLogic from LogicVision when the double-capture clocking scheme is used.

All these commercially available logic BIST products support the STUMPS-based architectures. Cadence supports a weighted-random spreading network (XOR network) for STUMPS with multiple-weight selects [Foote 1997]. For at-speed delay fault testing, ETLogic [LogicVision 2008] uses a **skewed-load-based at-speed BIST architecture**; TurboBIST-Logic [Wang 2005b, 2006b; SynTest 2008] implements the **double-capture-based at-speed BIST architecture**; and LBIST Architect [Mentor 2008] adopts a **hybrid at-speed BIST architecture** that supports both skewed-load and double-capture. In addition, all products provide inter-clock-domain delay fault testing for synchronous clock domains. On-chip clock controllers for testing these inter-clock-domain faults at-speed can be found in [Rajski 2003; Furukawa 2006; Nadeau-Dostie 2006, 2007; Keller 2007], and Table 3.7 summarizes the capture-clocking schemes for at-speed logic BIST that is used by the EDA vendors.

## 3.5 TEST COMPRESSION

***Test compression*** can provide 10× to 100× reduction or even more in the amount of test data (both test stimulus and test response) that must be stored on the ***automatic test equipment*** (ATE) [Touba 2006; Wang 2006a] for testing

**Table 3.7** Summary of Industry Practices for At-Speed Logic BIST

| Industry Practices | Skewed-Load | Double-Capture |
|---|---|---|
| Encounter test | Through service | Through service |
| ETLogic | √ | Through service |
| LBIST Architect | √ | √ |
| TurboBIST-Logic | | √ |

with a deterministic ATPG-generated test set. This greatly reduces ATE memory requirements and even more importantly reduces test time, because less data have to be transferred across the limited bandwidth between the ATE and the chip. Moreover, test compression methods are easy to adopt in industry because they are compatible with the conventional design rules and test generation flows used for scan testing.

Test compression is achieved by adding some additional on-chip hardware before the scan chains to decompress the test stimulus coming from the tester and after the scan chains to compact the response going to the tester. This is illustrated in Figure 3.37. This extra on-chip hardware allows the test data to be stored on the tester in a compressed form. Test data are inherently highly compressible because typically only 1% to 5% of the bits on a test pattern that is generated by an ATPG program have specified (*care*) values. Lossless compression techniques can thus be used to significantly reduce the amount of test stimulus data that must be stored on the tester. The on-chip **decompressor** expands the compressed test stimulus back into the original test patterns (matching in all the care bits) as they are shifted into the scan chains. The on-chip **compactor** converts long output response sequences into short signatures. Because the compaction is lossy, some fault coverage can be lost because

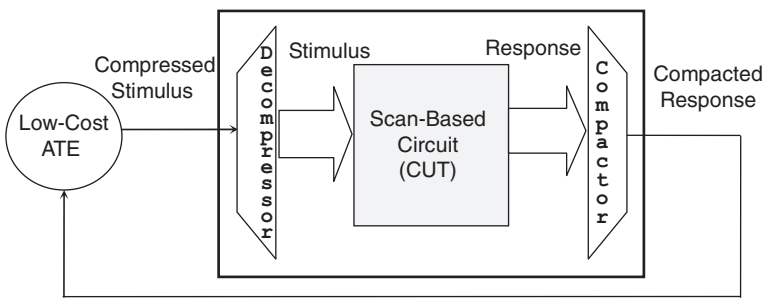



**FIGURE 3.37**

Architecture for test compression.

of unknown ($X$) values that might appear in the output sequence or aliasing where a faulty output response signature is identical to the fault-free output response signature. With proper design of the ***circuit under test*** (CUT) and the compaction circuitry, however, the fault coverage loss can be kept negligibly small.

## 3.5.1 Circuits for test stimulus compression

A **test cube** is defined as a deterministic test vector in which the bits that are not assigned values by the ATPG procedure are left as don't cares ($X$'s). Normally, ATPG procedures perform *random fill* in which all the $X$'s in the test cubes are filled randomly with 1's and 0's to create fully specified test vectors; however, for test stimulus compression, random fill is not performed during ATPG so the resulting test set consists of incompletely specified test cubes. The $X$'s make the test cubes much easier to compress than fully specified test vectors.

As mentioned earlier, test stimulus compression should be an information lossless procedure with respect to the specified (care) bits to preserve the fault coverage of the original test cubes. After decompression, the resulting test patterns shifted into the scan chains should match the original test cubes in all the specified (care) bits.

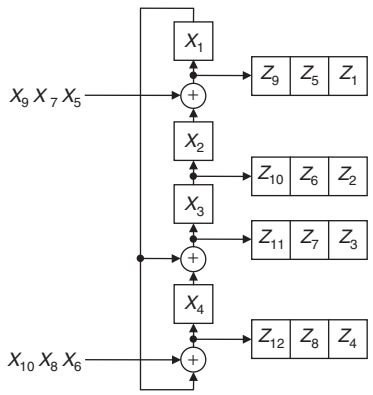
Many schemes for compressing test cubes have been surveyed in [Touba 2006; Wang 2006a]. Two schemes based on linear decompression and broadcast scan are described here in greater detail mainly because the industry has favored both approaches over code-based schemes from area overhead and compression ratio points of view. These code-based schemes can be found in [Wang 2006a].

### 3.5.1.1 *Linear-decompression-based schemes*

A class of test stimulus compression schemes is based on the use of **linear decompressors** to expand the data coming from the tester to fill the scan chains. Any decompressor that consists of only XOR gates and flip-flops is a ***linear decompressor*** [Könemann 1991]. Linear decompressors have a very useful property: their *output space* (*i.e.*, the space of all possible test vectors that they can generate) is a linear subspace that is spanned by a Boolean matrix. In other words, for any linear decompressor that expands an $m$-bit compressed stimulus from the tester into an $n$-bit stimulus (test vector), there exists a Boolean matrix $\boldsymbol{A}_{n\times m}$ such that the set of test vectors that can be generated by the linear decompressor is spanned by $\boldsymbol{A}$. A test vector $Z$ can be compressed by a particular linear decompressor if and only if there exists a solution to a system of linear equations, $\boldsymbol{A}X = Z$, where $\boldsymbol{A}$ is the **characteristic matrix** of the linear decompressor and $X$ is a set of **free variables** stored on the tester (every bit stored on the tester can be thought of as a "free variable" that can be assigned any value, 0 or 1).

The characteristic matrix for a linear decompressor can be obtained by symbolic simulation where each free variable coming from the tester is represented by a symbol. An example of this is shown in Figure 3.38, where a sequential linear decompressor containing an LFSR is used. The initial state of the LFSR is represented by free variables $X_1$ to $X_4$, and the free variables $X_5$ to $X_{10}$ are shifted in from two channels as the scan chains are loaded. After symbolic simulation, the final values in the scan chains are represented by the equations for $Z_1$ *to* $Z_{12}$. The corresponding system of linear equations for this linear decompressor is shown in Figure 3.39.

The symbolic simulation goes as follows. Assume that the initial seed $X_1$ *to* $X_4$ has been already loaded into the flip-flops. In the first clock cycle, the top flip-flop is loaded with the XOR of $X_2$ and $X_5$; the second flip-flop is loaded with $X_3$; the third flip-flop is loaded with the XOR of $X_1$ and $X_4$; and the bottom flip-flop is loaded with the XOR of $X_1$ and $X_6$. Thus, we obtain $Z_1 = X_2 \oplus X_5$, $Z_2 = X_3$, $Z_3 = X_1 \oplus X_4$, and $Z_4 = X_1 \oplus X_6$. In the second clock cycle, the top flip-flop is loaded with the XOR of the contents of the second flip-flop ($X_3$) and $X_7$; the second flip-flop is loaded with the contents of the third flip-flop ($X_1 \oplus X_4$); the third flip-flop is loaded with the XOR of the contents of the first flip-flop ($X_2 \oplus X_5$) and the fourth flip-flop ($X_1 \oplus X_6$); and the bottom flip-flop is loaded with the XOR of the contents of the first flip-flop ($X_2 \oplus X_5$) and $X_8$. Thus, we obtain $Z_5 = X_3 \oplus X_7$, $Z_6 = X_1 \oplus X_4$, $Z_7 = X_1 \oplus X_2 \oplus X_5 \oplus X_6$, and $Z_8 = X_2 \oplus X_5 \oplus X_8$. In the third clock cycle, the top flip-flop is loaded with




| | | |
|---|---|---|
| $Z_9 = X_1 \oplus X_4 \oplus X_9$ | $Z_5 = X_3 \oplus X_7$ | $Z_1 = X_2 \oplus X_5$ |
| $Z_{10} = X_1 \oplus X_2 \oplus X_5 \oplus X_6$ | $Z_6 = X_1 \oplus X_4$ | $Z_2 = X_3$ |
| $Z_{11} = X_2 \oplus X_3 \oplus X_5 \oplus X_7 \oplus X_8$ | $Z_7 = X_1 \oplus X_2 \oplus X_5 \oplus X_6$ | $Z_3 = X_1 \oplus X_4$ |
| $Z_{12} = X_3 \oplus X_7 \oplus X_{10}$ | $Z_8 = X_2 \oplus X_5 \oplus X_8$ | $Z_4 = X_1 \oplus X_6$ |

**FIGURE 3.38**

Example of symbolic simulation for linear decompressor.

$$\begin{pmatrix} 0&1&0&0&1&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0&0 \\ 1&0&0&1&0&0&0&0&0&0 \\ 1&0&0&0&0&1&0&0&0&0 \\ 0&0&1&0&0&0&1&0&0&0 \\ 1&0&0&1&0&0&0&0&0&0 \\ 1&1&0&0&1&1&0&0&0&0 \\ 0&1&0&0&1&0&0&1&0&0 \\ 1&0&0&1&0&0&0&0&1&0 \\ 1&1&0&0&1&1&0&0&0&0 \\ 0&1&1&0&1&0&1&1&0&0 \\ 0&0&1&0&0&0&1&0&0&1 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \\ X_8 \\ X_9 \\ X_{10} \end{pmatrix} = \begin{pmatrix} Z_1 \\ Z_2 \\ Z_3 \\ Z_4 \\ Z_5 \\ Z_6 \\ Z_7 \\ Z_8 \\ Z_9 \\ Z_{10} \\ Z_{11} \\ Z_{12} \end{pmatrix}$$

**FIGURE 3.39**

System of linear equations for the decompressor in Figure 3.38.

the XOR of the contents of the second flip-flop ($X_1 \oplus X_4$) and $X_9$; the second flip-flop is loaded with the contents of the third flip-flop ($X_1 \oplus X_2 \oplus X_5 \oplus X_6$); the third flip-flop is loaded with the XOR of the contents of the first flip-flop ($X_3 \oplus X_7$) and the fourth flip-flop ($X_2 \oplus X_5 \oplus X_8$); and the bottom flip-flop is loaded with the XOR of the contents of the first flip-flop ($X_3 \oplus X_7$) and $X_{10}$. Thus, we obtain $Z_9 = X_4 \oplus X_9$, $Z_{10} = X_1 \oplus X_6$, $Z_{11} = X_2 \oplus X_5 \oplus X_8$, and $Z_{12} = X_3 \oplus X_7 \oplus X_{10}$. At this point, the scan chains are fully loaded with a test cube, so the simulation is complete.

#### 3.5.1.1.1 Combinational linear decompressors

The simplest linear decompressors use only combinational XOR networks. Each scan chain is fed by the XOR of some subset of the channels coming from the tester [Bayraktaroglu 2001, 2003; Könemann 2003; Mitra 2006; Han 2007; Wang 2004, 2008]. The advantage compared with sequential linear decompressors is simpler hardware and control. The drawback is that, to encode a test cube, each **scan slice** (the $n$-bits that are loaded into the $n$ scan chains in each clock cycle) must be encoded with only the free variables that are shifted from the tester in a single clock cycle (which is equal to the number of channels). The worst-case most highly specified scan slices tend to limit the amount of compression that can be achieved, because the number of channels from the tester has to be sufficiently large to encode the most highly specified scan slices. Consequently, it is very difficult to obtain a high **encoding efficiency** (typically it will be less than 0.25); for the other less specified scan slices, a lot of the free variables end up getting wasted, because those scan slices could have been encoded with many fewer free variables.

One approach for improving the encoding efficiency of combinational linear decompressors that was proposed in [Krishna 2003] is to dynamically adjust the number of scan chains that are loaded in each clock cycle. So for a highly

specified scan slice, four clock cycles could be used in which 25% of the scan chains are loaded in each cycle, whereas for a lightly specified scan slice, only one clock cycle can be used in which 100% of the scan slices are loaded. This allows a better matching of the number of free variables with the number of specified bits to achieve a higher encoding efficiency. Note that it requires that the scan clock be divided into multiple domains.

#### 3.5.1.1.2 Sequential linear decompressors

**Sequential linear decompressors** are based on linear finite-state machines such as LFSRs, cellular automata, or ring generators [Mrugalski 2004]. The advantage of a sequential linear decompressor is that it allows free variables from earlier clock cycles to be used when encoding a scan slice in the current clock cycle. This provides much greater flexibility than combinational decompressors and helps avoid the problem of the worst-case most highly specified scan slices limiting the overall compression. The more flip-flops that are used in the sequential linear decompressor, the greater the flexibility that is provided. [Tobua 2006] classifies the sequential linear decompressors into two classes:

1. **Static reseeding** that computes a seed (an initial state) for each test cube [Touba 2006]. This seed, when loaded into an LFSR and run in autonomous mode, will produce the test cube in the scan chains [Könemann 1991]. This technique achieves compression by storing only the seeds instead of the full test cubes.
2. **Dynamic reseeding** calls for the injection of free variables coming from the tester into the LFSR as it loads the scan chains [Krishna 2001; Könemann 2001; Rajski 2004].

Figure 3.40 shows a generic example of a sequential linear decompressor that uses *b* channels from the tester to continuously inject free variables into the LFSR as it loads the scan chains through a combinational linear decompressor that typically is a combinational XOR network.

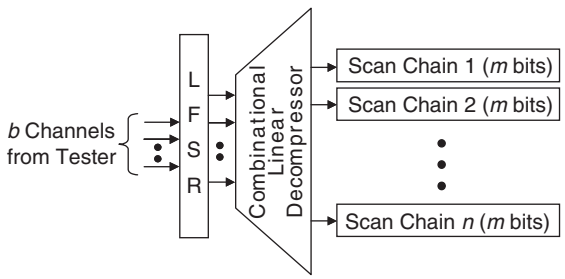



**FIGURE 3.40**

Typical sequential linear decompressor.

### 3.5.1.2 *Broadcast-scan-based schemes*

Another class of test stimulus compression schemes is based on broadcasting the same value to multiple scan chains. This was first proposed in [Lee 1998] and [Lee 1999]. Because of its simplicity and effectiveness, this method has been used as the basis of many test compression architectures, including some commercial *design for testability* (DFT) tools.

#### 3.5.1.2.1 Broadcast scan

To illustrate the basic concept of **broadcast scan**, first consider two independent circuits $C_1$ and $C_2$. Assume that these two circuits have their own test sets $T_1 = < t_{11}, t_{12}, \ldots, t_{1k} >$ and $T_2 = < t_{21}, t_{22}, \ldots, t_{2l} >$, respectively. In general, a test set may consist of random patterns and deterministic patterns. In the beginning of the ATPG process, usually random patterns are initially used to detect the easy-to-detect faults. If the same random patterns are used when generating both $T_1$ and $T_2$, then we may have $t_{11} = t_{21}$, $t_{12} = t_{22}$, ..., up to some $i$th pattern. After most faults have been detected by the random patterns, deterministic patterns are generated for the remaining difficult-to-detect faults. Generally, these patterns have many "don't care" bits. For example, when generating $t_{1(i+1)}$, many "don't care" bits may still exist when no more faults in $C_1$ can be detected. By use of a test pattern with bits assigned so far for $C_1$, we can further assign specific values to the "don't care" bits in the pattern to detect faults in $C_2$. Thus, the final pattern would be effective in detecting faults in both $C_1$ and $C_2$.

The concept of pattern sharing can be extended to multiple circuits as illustrated in Figure 3.41. One major advantage of the use of *broadcast scan* for independent circuits is that all faults that are detectable in all original circuits will also be detectable with the broadcast structure. This is because if one test vector can detect a fault in a stand-alone circuit, then it will still be possible to apply this vector to detect the fault in the broadcast structure. Thus, the broadcast scan method will not affect the fault coverage if all circuits are independent. Note that broadcast scan can also be applied to multiple scan chains of a single circuit if all subcircuits driven by the scan chains are independent.

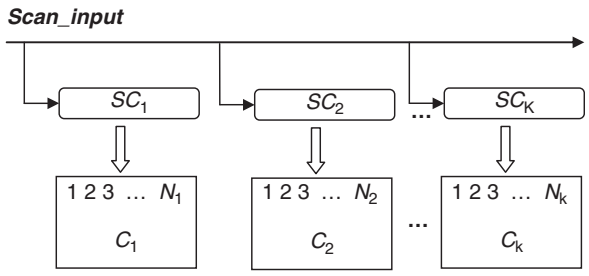



**FIGURE 3.41**

Broadcasting to scan chains driving independent circuits.

#### 3.5.1.2.2 Illinois scan

If *broadcast scan* is used for multiple scan chains of a single circuit where the subcircuits driven by the scan chains are not independent, then the property of always being able to detect all faults is lost. The reason for this is that if two scan chains are sharing the same channel, then the *i*th scan cell in each of the two scan chains will always be loaded with identical values. If some fault requires two such scan cells to have opposite values to be detected, it will not be possible to detect this fault with broadcast scan.

To address the problem of some faults not being detected when broadcast scan is used for multiple scan chains of a single circuit, the **Illinois scan architecture** was proposed in [Hamzaoglu 1999] and [Hsu 2001]. This scan architecture consists of two modes of operations, namely a *broadcast mode* and a *serial scan mode,* which are illustrated in Figure 3.42. The *broadcast mode* is first used to detect most faults in the circuit. During this mode, a scan chain is divided into multiple subchains called *segments,* and the same vector can be shifted into all segments through a single shared scan-in input. The response data from all subchains are then compacted by a MISR or other space/time compactor. For the remaining faults that cannot be detected in broadcast mode, the *serial scan mode* is used where any possible test pattern can be applied. This ensures that complete fault coverage can be achieved. The extra logic required to implement the Illinois scan architecture consists of several multiplexers and some simple control logic to switch between the two modes. The area overhead of this logic is typically quite small compared with the overall chip area.

The main drawback of the Illinois scan architecture is that no test compression is achieved when it is run in *serial scan mode*. This can significantly degrade the overall **compression ratio** if many test patterns must be applied in serial scan mode. To reduce the number of patterns that need to be applied in serial scan mode, multiple-input broadcast scan or reconfigurable broadcast scan can be used. These techniques are described next.

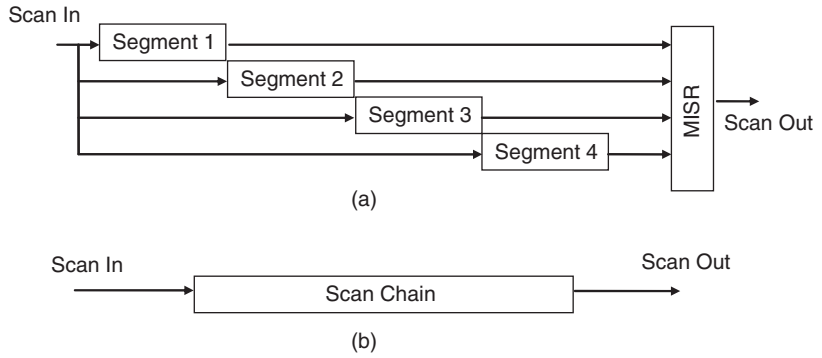



**FIGURE 3.42**

Two modes of Illinois scan architecture: (a) Broadcast mode. (b) Serial scan mode.

#### 3.5.1.2.3 Multiple-input broadcast scan

Instead of the use of only one channel to drive all scan chains, a **multiple-input broadcast scan** could be used where there is more than one channel [Shah 2004]. Each channel can drive some subset of the scan chains. If two scan chains must be independently controlled to detect a fault, then they could be assigned to different channels. The more channels that are used and the shorter each scan chain is, the easier to detect more faults because fewer constraints are placed on the ATPG. Determining a configuration that requires the minimum number of channels to detect all detectable faults is thus highly desired with a multiple-input broadcast scan technique.

#### 3.5.1.2.4 Reconfigurable broadcast scan

*Multiple-input broadcast scan* may require a large number of channels to achieve high fault coverage. To reduce the number of channels that are required, a **reconfigurable broadcast scan** method can be used. The idea is to provide the capability to reconfigure the set of scan chains that each channel drives. Two possible reconfiguration schemes have been proposed, namely **static reconfiguration** [Pandey 2002; Wang 2002; Samaranayake 2003; Chandra 2007], and **dynamic reconfiguration** [Li 2004; Sitchinava 2004; Wang 2004, 2008; Mitra 2006; Wohl 2007a]. In *static reconfiguration*, the reconfiguration can only be done when a new pattern is to be applied. For this method, the target fault set can be divided into several subsets, and each subset can be tested by a single configuration. After testing one subset of faults, the configuration can be changed to test another subset of faults. In *dynamic reconfiguration*, the configuration can be changed while scanning in a pattern. This provides more reconfiguration flexibility and hence can, in general, lead to better results with fewer channels. This is especially important for hard cores, when the test patterns provided by core vendor cannot be regenerated. The drawback of dynamic reconfiguration *versus* static reconfiguration is that more control information is needed for reconfiguring at the right time, whereas for static reconfiguration the control information is much less because the reconfiguration is done only a few times (only after all the test patterns that use a particular configuration have been applied).

Figure 3.43 shows an example *multiplexer* (MUX) network that can be used for dynamic configuration. When a value on the control line is selected, particular data at the four input pins are broadcasted to the eight scan chain inputs. For instance, when the control line is set to 0 (or 1), the scan chain 1 output will receive input data from Pin 4 (or Pin 1) directly.

#### 3.5.1.2.5 Virtual scan

Rather than the use of MUX networks for test stimulus compression, combinational logic networks can also be used as decompressors. The combinational logic network can consist of any combination of simple combinational gates, such as buffers, inverters, AND/OR gates, MUXs, and XOR gates. This scheme, referred to as **virtual scan**, is different from *reconfigurable broadcast scan* and

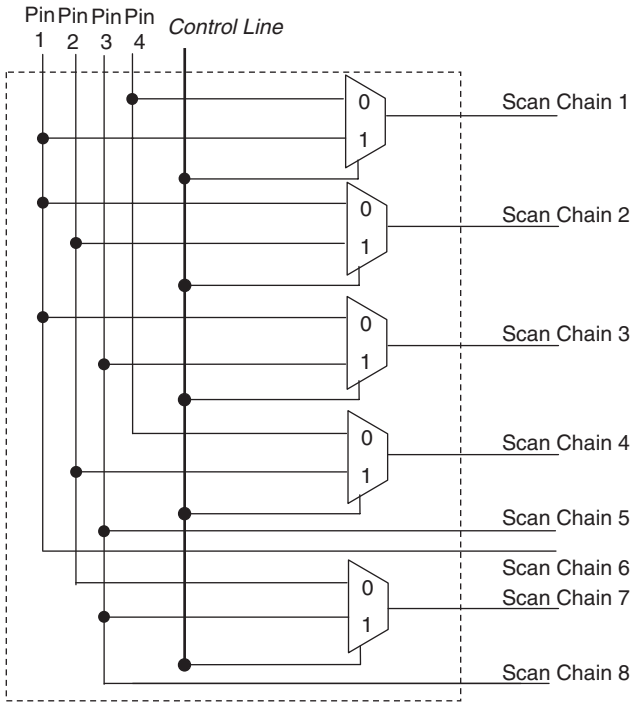



**FIGURE 3.43**

Example MUX network with control line(s) connected only to select pins of the multiplexers.

*combinational linear decompression* where pure MUX and XOR networks are allowed, respectively. The combinational logic network and the order of the scan chains can be specified as a set of constraints or just as an expanded circuit for ATPG. In either case, the test cubes that ATPG generates are the compressed stimuli for the decompressor itself. There is no need to solve a system of linear equations, and *dynamic compaction* can be effectively used during the ATPG process. Hence, only one-pass ATPG is required during test stimulus compression.

The *virtual scan* scheme was proposed in [Wang 2002, 2004, 2008]. In these papers, the decompressor was referred to as a **broadcaster**. The authors also proposed adding additional logic, when required, through *VirtualScan inputs* to reduce or remove the constraints imposed by the broadcaster on the circuit, thereby yielding very little or no fault coverage loss caused by test stimulus compression. For instance, a **scan connector** consisting of a set of multiplexers that places scan cells in the scan chains in a particular order can be connected to the outputs of the combinational logic network during each virtual scan test mode. Because the scan chains are reordered in each test mode, the imposed constraints of the combinational logic network on the circuit are reduced or removed.

In a broad sense, *virtual scan* is a generalized class of broadcast scan, Illinois scan, multiple-input broadcast scan, reconfigurable broadcast scan, and combinational linear decompression. The advantage of the use of virtual scan is that it allows the ATPG to directly search for a test cube that can be applied by the decompressor and allows very effective dynamic compaction. Thus, virtual scan may produce shorter test sets than any test stimulus compression scheme based on solving linear equations; however, because this scheme may impose XOR or MUX constraints directly on the original circuit, it may take longer than those based on solving linear equations to generate test cubes or compressed stimuli. Two example virtual scan decompression circuits are shown in Figures 3.44a and 3.44b, respectively [Wang 2008]. Additional VirtualScan inputs are used to further reduce the XOR or MUX constraints imposed on the original circuit. An XOR network similar to the broadcaster shown in Figure 3.44a is sometimes referred to as a **space expander** or a **spreading network** in logic BIST applications.

### 3.5.2 Circuits for test response compaction

Test response compaction is performed at the outputs of the scan chains. The purpose is to reduce the amount of test response that needs to be transferred back to the tester. Although test stimulus compression must be lossless, test response compaction can be lossy. A large number of different test response compaction schemes and associated (response) compactors have been presented in the literature [Wang 2006a]. The effectiveness of each compaction scheme and the chosen compactor depends on its ability to avoid *aliasing* and tolerate *unknown test response bits* or *X*'s. These schemes can be grouped into three categories: (1) **space compaction**, (2) **time compaction**, and (3) **mixed space and time compaction**.

A **space compactor** compacts an $m$-bit-wide output pattern to an $n$-bit-wide output pattern (where $n < m$). A **time compactor** compacts $p$ output patterns to $q$ output patterns (where $q < p$). A **mixed space and time compactor** has both space and time compaction performed concurrently. Typically, a space compactor is composed of XOR gates [Saluja 1983]; a time compactor includes a *multiple-input signature register* (MISR) [Frohwerk 1977]; and a mixed space and time compactor adds a space compactor at either the input or the output side of a time compactor [Saluja 1983; Wohl 2001]. Because test response compaction can be combinational-logic-based or sequential-logic-based, without loss of generality, we refer space compaction to as a **combinational compaction** scheme, and time compaction as well as mixed space and time compaction to as **sequential compaction** schemes.

There are three sources of aliasing according to [Wohl 2001]: (1) **combinational cancellation** occurs when two or more erroneous scan chain outputs (compactor inputs) are XORed in the compactor during the same cycle, which

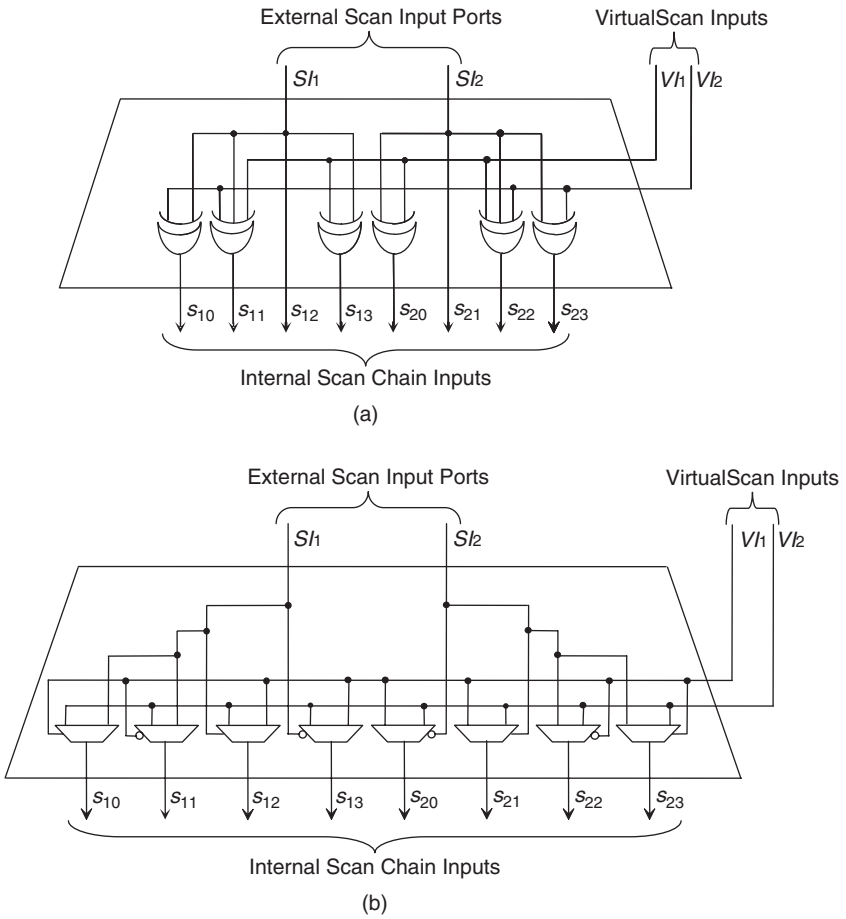



**FIGURE 3.44**

Example virtual scan decompression circuits: (a) Broadcaster that sees an example XOR network with additional VirtualScan inputs to reduce coverage loss. (b) Broadcaster that uses an example MUX network with additional VirtualScan inputs that can be also connected to data pins of the multiplexers.

cancel out the error effects in that cycle; (2) **shift cancellation** occurs when one or more erroneous scan chain output bits captured into the compactor are cancelled out by other erroneous scan chain output bits when the former are shifted down the shift path of the compactor; and (3) **feedback cancellation** occurs when one or more errors captured into the compactor during one cycle propagate through some feedback path of the compactor and cancel out with errors in later cycles. Combinational cancellation will exist in space compaction as well as mixed space and time compaction, because *non-aliasing*

*space compactors* are impractical for real designs [Chakrabarty 1998; Pouya 1998]. On the other hand, shift cancellation and feedback cancellation are only present when either time compaction or mixed space and time compaction is used; however, shift cancellation is independent of the compactor feedback structure and its polynomial, whereas feedback cancellation depends on the compactor polynomial chosen.

Because unknown test response bits (*X*'s) can potentially reduce the fault coverage of the circuit under test when a combinational compactor is used and corrupt the final signature in a sequential compactor, one safe approach is to completely block these *X*'s before they reach the **response compactor** (combinational compactor or sequential compactor). During design, these potential **X-generators** (**X-sources**) can be identified with a scan design rule checker. When the *X* effects of an X-generator are likely to reach the response compactor, these *X*'s must be blocked before they reach the compactor [Gu 2001]. The process is often referred to as **X-blocking** or **X-bounding**.

In X-blocking, an X-source can be blocked either at the X-source or anywhere along its propagation paths before *X*'s reach the compactor. In case the X-source has been blocked at a nearby location during test and will not reach the compactor, there is no need to block the X-source; however, care must be taken to ensure that no observation points are added between the X-source and the location at which it is blocked to avoid capturing potential *X*'s into the compactor.

A simple example illustrating the X-blocking scheme for an X-source is shown in Figure 3.45. The output of the X-source is blocked and forced to 0 by setting the *select* signal of the multiplexer (MUX) to a fixed value (selecting the 0 input) in test mode. As a separate example, a non-scan flip-flop that is neither scanned nor initialized is a potential X-generator (X-source). If the flip-flop has two outputs (*Q* and *QB*), one can add two multiplexers forcing both outputs to opposite values in test mode. Alternately, if the flip-flip has an asynchronous set/reset pin, an AND/OR control point can be added to permanently force the flip-flip to 0 or 1 during test. Although an AND/OR control point can be added to force the non-scan flip-flop to a constant value, it is recommended that for

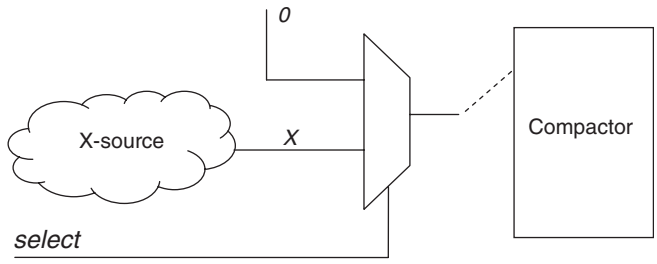



**FIGURE 3.45**

A simple illustration of the X-blocking scheme.

better fault coverage inserting a MUX control point driven by a nearby existing scan cell is preferred.

X-blocking can ensure that no *X*'s will be propagated to the compactor; however, it also blocks the fault effects that can only propagate to an observable point through the now-blocked X-source (*e.g.*, the non-scan flip-flop). This can result in fault coverage loss. This problem can be addressed by use of a more flexible control on the *select* signal such that the X-source is blocked only during the cycles at which it may generate *X*'s. Alternately, if the number of such faults for a given bounded X-generator justifies the cost, one or more observation points can be added before the X-source (*e.g.*, at the *D* input of the non-scan flip-flop) to provide an observable point to which those faults can propagate. These X-blocking or X-bounding methods have been extensively discussed in [Wang 2006a].

In this subsection, we only present some compactor designs that are widely used in industry along with some emerging compactors. For more information, refer to the key references cited in [Patel 2003; Mitra 2004b; Rajski 2004; Volkerink 2005; Wang 2006a; Touba 2007; Wohl 2007b].

#### 3.5.2.1 *Combinational compaction*

A combinational compactor uses a combinational circuit to compact $m$ outputs of the circuit under test into $n$ test outputs, where $n < m$. If each output sequence contains only known (non-*X*) values (0's and 1's), then a combinational compactor that uses XOR gates with each internal scan chain output connected to only one XOR gate input is sufficient to guarantee no-fault coverage loss when the number of errors appearing at the $m$ outputs is always odd [Saluja 1983]. A compactor that uses such XOR gates is referred to as a **conventional combinational compactor** or **simple space compactor**. An example is illustrated in Figure 3.46 [Wang 2008]. On the contrary, if any output sequence contains unknown values (*X*'s), the combinational compaction scheme must have the capability to mask off or tolerate unknowns to prevent faults from going undetected. A compactor able to mask off or tolerate *X*'s is referred to as an **X-tolerant combinational compactor** or **X-tolerant space compactor**. Two representative schemes currently practiced in industry are discussed in the following: (1) X-compact and (2) X-impact. Other schemes to further tolerate the amount of *X*'s can be found in [Patel 2003; Rajski 2004; Wohl 2004, 2007b; Wang 2008].

##### 3.5.2.1.1 X-compact

**X-compact** [Mitra 2004a] is an **X-tolerant space compaction** technique that connects each internal scan chain output to two or more external scan output ports through a network of XOR gates to tolerate unknowns. A response compaction circuit designed by use of the X-compact technique is called an **X-compactor**. Figure 3.47 shows an X-compactor with eight inputs and five outputs. It is composed of four 3-input XOR gates and eleven 2-input XOR gates.

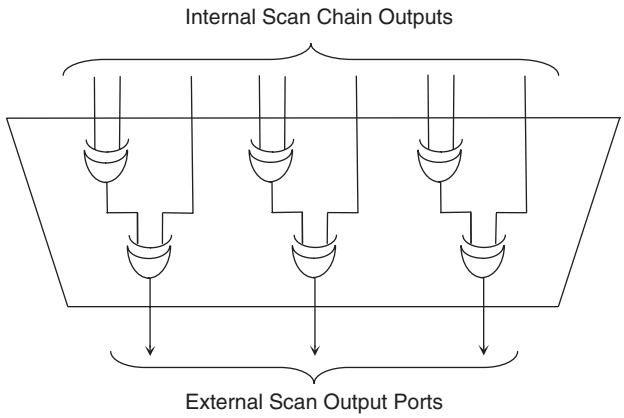



**FIGURE 3.46**

A conventional combinational compactor with nine inputs and three outputs.

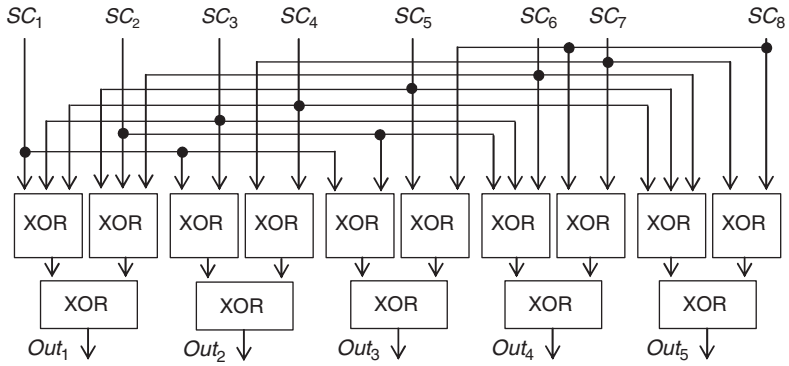



**FIGURE 3.47**

An X-compactor with eight inputs and five outputs.

Only one aliasing source, namely *combinational cancellation*, can exist in an X-compactor because of its combinational property. As an extreme example, if an X-compactor has only one output, it is, indeed, a **parity checker**, and any two error bits occurring simultaneously from the internal scan chain outputs will lead to aliasing.

Although aliasing may still exist when the X-compact technique is used, one can design an X-compactor that guarantees zero-aliasing in many practical cases. Consider Figure 3.47 again. If only one error bit occurs at the *SC* inputs, the error will be propagated to some output of the compactor and thus detected. One can also find that the compactor can detect any two or any odd number of errors that occur at the same cycle. In the following we use a binary matrix, called an **X-compact matrix**, to represent an X-compactor and to illustrate the fault detectability and X-tolerability of the compactor.

Suppose that the outputs of $m$ scan chains are to be compacted into $n$ bits for each scan cycle with an X-compactor. The associated *X-compact matrix* then contains $n$ rows and $k$ columns, in which each row corresponds to a scan chain output (*e.g.*, *SC* in Figure 3.47), and each column corresponds to an X-compactor output (*e.g.*, *Out* in Figure 3.47). The entry at row $i$ and column $j$ of the matrix is 1 if and only if the $j$th X-compactor output depends on the $i$th scan chain output; otherwise, the matrix entry is 0. Thus, the corresponding X-compact matrix $M$ of the X-compactor shown in Figure 3.47 is:

$$M = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

With the help of an X-compact matrix, it was shown in [Mitra 2004a] that errors from any one, two, or an odd number of scan chains at the same scan-out cycle are guaranteed to be detected by an X-compactor if every row of the corresponding X-compact matrix of the compactor is distinct and contains an odd number of 1's. This can be proved by the observation that (1) if all rows of the X-compact matrix are distinct, then a bitwise XOR of any two rows is nonzero, and (2) if each row further contains an odd number of 1's, then the bitwise XOR of any odd number of rows also contains an odd number of 1's.

The most distinctive feature of the X-compact technique is its X-tolerant capability (*i.e.*, detecting error bits even when the scan chain outputs have unknown bits). Refer to Figure 3.47 again. If one unknown bit occurs at $SC_1$, then the unknown value will be spread to $Out_1$, $Out_2$, and $Out_3$. Thus, after the XOR operation, the values at $Out_1$, $Out_2$, and $Out_3$ are masked (becoming unknown). However, if there is only one error bit in all other scan chain outputs, then the error bit will still be detected, because the error bit will be spread to at least one output that is not $Out_1$, $Out_2$, or $Out_3$. For example, an error bit occurring at $SC_2$ will be detected from $Out_4$. Thus, we have the following X-tolerant theorem:

*Theorem 3.1:*
An error from any scan chain with one unknown bit from any other scan chain at the same cycle is guaranteed to be observed at the outputs of an X-compactor if and only if:

1. No row of the X-compact matrix contains all 0's.
2. For any X-compact matrix row, the submatrix obtained by removing the row responding to the scan chain output with unknown bit and all columns having 1's in that row does not contain a row with all 0's.

The X-compact matrix of Figure 3.47 satisfies the preceding theorem. For example, if we remove row 1 and columns 1, 2, and 3, then each of the remaining rows in the submatrix contains at least a 1. Theorem 3.1 can be further extended to deal with errors from any $k_1$ or fewer scan chains with unknown bits from any $k_2$ or fewer scan chains ($k_1 + k_2 \leq n$) as follows:

*Theorem 3.2:*
Errors from any $k_1$ or fewer scan chains with unknown bits from any $k_2$ or fewer scan chains at the same cycle, where $k_1 + k_2 \leq n$ and $n$ is the number of scan chains, are guaranteed to be observed at the outputs of an X-compactor if and only if:

1. No row of the X-compact matrix contains all 0's.
2. For any set $S$ of $k_1$ X-compact matrix rows, any set of $k_2$ rows in the submatrix obtained by removing the rows in $S$ and the X-compact matrix columns having 1's in the rows in $S$ are linearly independent.

Designing an X-compact matrix to satisfy Theorem 3.2 is a complicated problem when an X-compactor is expected to tolerate three or more unknown bits. In some cycles, the number of actual knowns appearing at the scan chain outputs could exceed the number of unknowns designed to be tolerated by the X-compactor. Hence, the fault detectability and X-tolerability of an X-compactor highly depends on its actual implementation and the number of unknowns to be tolerated.

#### 3.5.2.1.2 X-impact

Although X-blocking and X-compact each can achieve significant reduction in fault coverage loss caused by *X*'s present at the inputs of a combinational compactor, the **X-impact** technique described in [Wang 2004] is helpful in that it can further reduce fault coverage loss simply by use of ATPG to algorithmically handle the impact of residual *X*'s on the combinational compactor without adding any extra circuitry. The combinational compactor in use can be either a conventional combinational compactor or an X-tolerant combinational compactor.

**Example 3.1** An example of algorithmically handling X-impact is shown in Figure 3.48. Here, $SC_1$ to $SC_4$ are scan cells connected to a conventional combinational compactor composed of XOR gates $G_7$ and $G_8$. Lines $a$, $b$, ..., $h$ are internal signals, and line $f$ is assumed to be connected to an X-source (memory, non-scan storage element, etc.). Now consider the detection of the stuck-at-0 (SA0) fault $f_1$. Logic value 1 should be assigned to both lines $d$ and $e$ to activate $f_1$. The fault effect will be captured by scan cell $SC_3$. If the $X$ on $f$ propagates to $SC_4$, then the compactor output q will become $X$ and $f_1$ cannot be detected. To avoid this, ATPG can try to assign either 1 to line $g$ or 0 to line h to block the $X$ from reaching $SC_4$. If it is impossible to achieve this assignment, ATPG can then try to assign 1 to line $c$, 0 to line $b$, and 0 to line $a$ to propagate the fault effect to $SC_2$. As a result, fault $f_1$ can be detected. Thus, X-impact is avoided by algorithmic assignment without adding any extra circuitry.

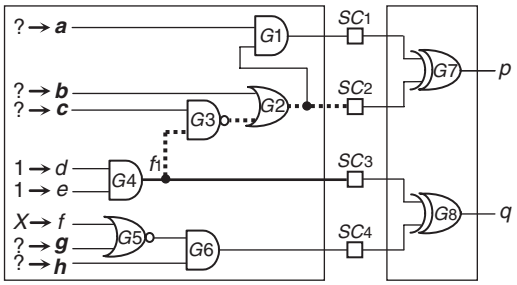



**FIGURE 3.48**

Handling of X-impact.

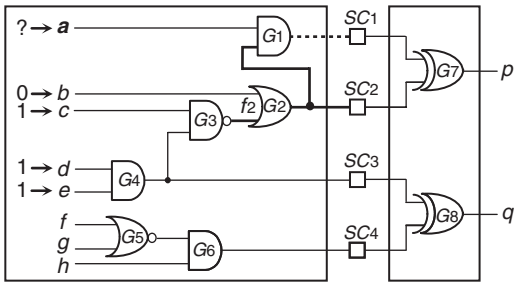



**FIGURE 3.49**

Handling of aliasing.

**Example 3.2** It is also possible to use the X-impact approach to reduce combinational cancellation (an aliasing source). An example of algorithmically handling aliasing is shown in Figure 3.49. Here, $SC_1$ to $SC_4$ are scan cells connected to a conventional combinational compactor composed of XOR gates $G_7$ and $G_8$. Lines $a$, $b$, . . . , $h$ are internal signals. Now consider the detection of the stuck-at-1 fault $f_2$. Logic value 1 should be assigned to lines $c$, $d$, and e to activate $f_2$, and logic value 0 should be assigned to line b to propagate the fault effect to $SC_2$. If line a is set to 1, then the fault effect will also propagate to $SC_1$. In this case, aliasing will cause the compactor output p to have a fault-free value, resulting in an undetected $f_2$. To avoid this, ATPG can try to assign 0 to line a to block the fault effect from reaching $SC_1$. As a result, fault $f_2$ can be detected. Thus, aliasing can be avoided by algorithmic assignment without any extra circuitry.

### 3.5.2.2 *Sequential compaction*

In contrast to a combinational compactor that typically uses XOR gates to compact output responses, a sequential compactor uses sequential logic instead. The sequential compactor can be a **time-space compressor** or a **space-time compressor** as described in [Saluja 1983], although the authors only considered output bit streams of 0's and 1's. The type of sequential logic to be used

for response compaction depends on whether the output responses contain unknown values (*X*'s). A sequential compactor capable of masking off or tolerating these *X*'s is often referred to as an **X-tolerant sequential compactor**.

#### 3.5.2.2.1 Signature analysis

If X-bounding as described previously has been used such that each output response does not contain any unknown (*X*) values, then the *multiple-input signature register* (MISR) widely used for logic BIST applications can be simply used [Frohwerk 1977]. Referred to as a **conventional sequential compactor**, the MISR uses an XOR gate at each MISR stage input to compact the output sequences, $M_0$ to $M_3$, into the *linear feedback shift register* (LFSR) simultaneously. The final contents stored in the MISR after compaction is often called the (*final*) *signature* of the MISR. A conventional sequential compactor that uses a four-stage MISR is illustrated in Figure 3.50. For more information on signature analysis and the MISR design, the reader is referred to Section 3.4.2.3.

#### 3.5.2.2.2 X-masking

On the contrary, if the output response contains unknown (*X*) values, then one must make sure when the sequential compactor is used that no *X*'s from the circuit under test will reach the compactor. Although it may not result in fault coverage loss, the X-bounding scheme described previously does add area overhead and may impact delay because of the inserted logic. It is not surprising to find that, in complex designs, more than 25% of scan cycles could contain one or more *X*'s in the test response. It is difficult to eliminate these residual *X*'s by DFT; thus, an encoder with high X-tolerance is very attractive. Instead of blocking the *X*'s where they are generated, the *X*'s can also be masked off right before the sequential compactor. This scheme is referred to as **X-masking**. A typical X-masking circuit is shown in Figure 3.51. The mask controller applies a logic value 1 at the appropriate time to mask off any scan output that contains an *X* before the *X* reaches the compactor.

The **X-masking compactor** is one type of X-tolerant sequential compactors. Typically, it implies that sequential logic (comprising one or more MISRs or SISRs) is used in the compactor for response compaction. Almost all existing X-tolerant sequential compactors proposed in the literature use X-masking, including OPMISR+ [Barnhart 2002; Naruse 2003], ETCompression [Nadeau-Dostie 2004],

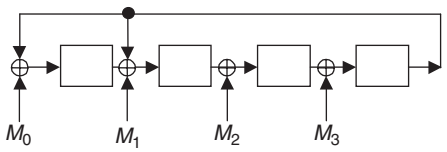



**FIGURE 3.50**

A conventional sequential compactor that uses a four-stage MISR.

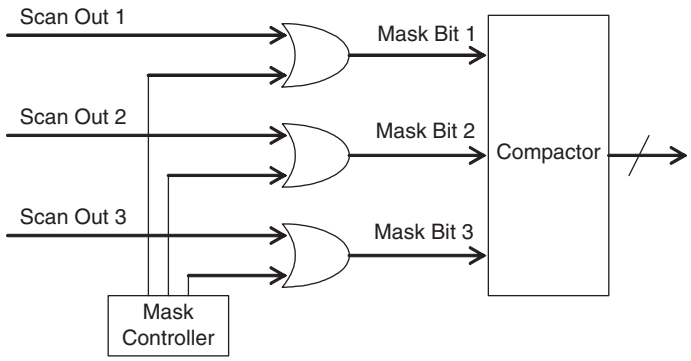



**FIGURE 3.51**

An example X-masking circuit in use with a compactor.

and convolutional compactors [Mitra 2004b; Rajski 2005, 2008]. In fact, combinational logic (such as XOR gates) can also be used in the compactor. Such an X-masking compactor that uses combinational logic is referred to as a **selective compactor** [Rajski 2004]. Mask data are needed to indicate when the masking should take place. These mask data can be stored in compressed format and can be decompressed with on-chip hardware. Possible compression techniques are *weighted pseudo-random LFSR reseeding* or *run-length encoding* [Volkerink 2005].

Another type of X-tolerant sequential compactor is an **X-canceling MISR** [Touba 2007, 2008] that does not mask the *X*'s before they enter the MISR. It allows the *X*'s to be compacted in a MISR and then selectively XORs together combinations of MISR signature bits that are linearly dependent in terms of the *X*'s such that all the *X*'s are canceled out.

#### 3.5.2.2.3 *q*-compact

In case none of the X-bounding, X-masking, or X-canceling schemes is available to block, mask off, or cancel all *X*'s, the sequential logic in use must not have a feedback path so these *X*'s will only stay in the sequential compactor for a few clock cycles. Such an X-tolerant sequential compaction scheme is referred to as ***q*-compact**. A ***q*-compactor** that uses this X-tolerant compaction scheme is illustrated in [Han 2006].

Figure 3.52 shows an example of a *q*-compactor assuming the inputs are coming from internal scan chain outputs [Han 2006]. The spatial part of the *q*-compactor consists of single-output XOR networks (called *spread networks*) connected to the flip-flops by means of additional 2-input XOR gates interspersed between successive storage elements. As can be seen, every error in a scan cell can reach storage elements and then outputs in several possible ways. The spread network that determines this property is defined in terms of

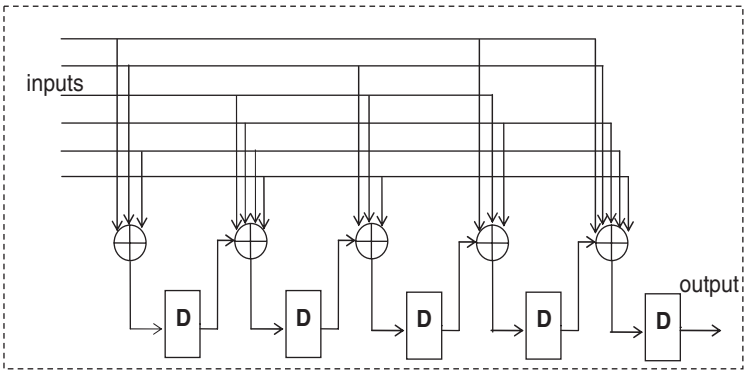



**FIGURE 3.52**

An example *q*-compactor with single output.

*spread polynomials* indicating how particular scan chains are connected to the register flip-flops.

Different from a conventional MISR, the *q*-compactor presented in Figure 3.52 does not have a feedback path; consequently, any error or *X* injected into the compactor is shifted out after at most five clock cycles. The shifted-out data will be compared with the expected data and then the error will be detected.

### 3.5.3 **Industry practices**

Several test compression products and solutions have been introduced by some of the major DFT vendors in the CAD industry. These products differ significantly with regard to technology, design overhead, design rules, and the ease of use and implementation. A few second-generation products have also been introduced by a few of the vendors [Kapur 2008]. This subsection summarizes a few of the products introduced by companies such as Cadence Design Systems [Cadence 2008], LogicVision [LogicVision 2008], Mentor Graphics [Mentor 2008], Synopsys [Synopsys 2008], and SynTest Technologies [SynTest 2008].

Current industry solutions can be grouped under two main categories for stimulus decompression. The first category uses *linear-decompression–based schemes*, whereas the second category uses *broadcast-scan–based schemes*. The main difference between the two categories is the manner in which the ATPG engine is used. The first category includes products, such as ETCompression [LogicVision 2008] from LogicVision, TestKompress [Rajski 2004] from Mentor Graphics, XOR Compression [Cadence 2008] from Cadence, and SOCBIST [Wohl 2003] from Synopsys. The second category includes products, such as OPMISR+ [Barnhart 2002; Cadence 2008] from Cadence, VirtualScan [Wang 2004, 2008] from SynTest, and DFT MAX [Sitchinava 2004; Wohl 2007a] from Synopsys.

For designs that use *linear-decompression–based schemes*, test compression is achieved in two distinct steps. During the first step, conventional ATPG is used to generate sparse ATPG patterns (called test cubes), in which *dynamic compaction* is performed in a nonaggressive manner, while leaving unspecified bit locations in each test cube as *X*. This is accomplished by not aggressively performing the *random fill* operation on the test cubes, which is used to increase coverage of individual patterns, and hence reduce the total pattern count. During the second step, a system of linear equations, describing the hardware mapping from the external scan input ports to the internal scan chain inputs, are solved to map each test cube into a compressed stimulus that can be applied externally. If a mapping is not found, a new attempt at generating a new test cube is required.

For designs that use *broadcast-scan–based schemes*, only a single step is required to perform test compression. This is achieved by embedding the constraints introduced by the decompressor as part of the ATPG tool, such that the tool operates with much more restricted constraints. Hence, whereas in conventional ATPG, each individual scan cell can be set to 0 or 1 independently, for *broadcast-scan–based schemes* the values to which related scan cells can be set are constrained. Thus, a limitation of this solution is that in some cases, the constraints among scan cells can preclude some faults from being tested. These faults are typically tested as part of a later top-up ATPG process if required, similar to the use of *linear-decompression–based schemes*.

On the response compaction side, industry solutions have used either combinational compactors such as XOR networks, or sequential compactors such as MISRs, to compact the test responses. At present, combinational compactors have a higher acceptance rate in the industry because they do not involve the process of guaranteeing that no unknown (*X*) values are generated in the circuit under test.

A summary of the different compression architectures used in the commercial products is shown in Table 3.8. Six products from five DFT companies are included. Since June 2006, Cadence has added XOR Compression as an alternative to the OPMISR+ product described in [Wang 2006a].

**Table 3.8** Summary of Industry Practices for Test Compression

| Industry Practices | Stimulus Decompressor | Response Compactor |
|---|---|---|
| XOR Compression or OPMISR+ | Combinational XOR Network or Fanout Network | XOR Network with or without MISR |
| TestKompress | Ring Generator | XOR Network |
| VirtualScan | Combinational Logic Network | XOR Network |
| DFT MAX | Combinational MUX Network | XOR Network |
| ETCompression | (Reseeding) PRPG | MISR |

**Table 3.9** Summary of Industry Practices for At-Speed Delay Fault Testing

| Industry Practices | Skewed-Load | Double-Capture |
|---|---|---|
| XOR Compression or OPMISR+ | √ | √ |
| TestKompress | √ | √ |
| VirtualScan | √ | √ |
| DFT MAX | √ | √ |
| ETCompression | √ | Through Service |

It is evident that the solutions offered by the current EDA DFT vendors are quite diverse with regard to stimulus decompression and response compaction. For stimulus decompression, OPMISR+, VirtualScan, and DFT MAX are broadcast-scan–based, whereas TestKompress and ETCompression are linear-decompression–based. For response compaction, OPMISR+ and ETCompression can include MISRs, whereas four other solutions purely adopt (X-tolerant) XOR networks. What is common is that all six products provide their own diagnostic solutions.

Generally speaking, any modern ATPG compression program supports at-speed clocking schemes used in its corresponding at-speed scan architecture. For at-speed delay fault testing, ETCompression currently uses a **skewed-load–based at-speed test compression architecture** for ATPG. The product can also support the double-capture clocking scheme through service. All other ATPG compression products, including OPMISR+, TestKompress, VirtualScan, and DFT MAX, support the **hybrid at-speed test compression architecture** by use of both skewed-load (*a.k.a.* launch-on-shift) and double-capture (*a.k.a.* launch-on-capture). In addition, almost every product supports inter-clock-domain delay fault testing for synchronous clock domains. A few on-chip clock controllers for detecting these inter-clock-domain delay faults at-speed have been proposed in [Beck 2005; Nadeau-Dostie 2005, 2006; Furukawa 2006; Fan 2007; and Keller 2007].

The clocking schemes used in these commercial products are summarized in Table 3.9. It should be noted that compression schemes might be limited in effectiveness if there are a large number of unknown response values, which can be exacerbated during at-speed testing when many paths do not make the timing being used.

## 3.6 CONCLUDING REMARKS

*Design for testability* (DFT) has become vital for ensuring circuit testability and product quality. Scan design, which has proven to be the most powerful DFT technique ever invented, allowed the transformation of sequential circuit testing into

combinational circuit testing and has since become an industry standard. Currently, a scan design can contain a billion transistors [Naffziger 2006; Stackhouse 2008]. To screen all possible physical failures (manufacturing defects) caused by manufacturing imperfection, test compression coupled to scan design has rapidly emerged, becoming a crucial DFT technique to address the explosive test data volume and long test application time problems. At the same time, scan-based logic *built-in self-test* (BIST) is of growing importance because of its inherent advantage of performing self-test on-chip, on-board, or in-system, which can substantially improve the reliability of the system and the ability of in-field diagnosis.

Whereas the STUMPS-based architecture [Bardell 1982] is the most popular logic BIST architecture practiced currently for scan-based designs, the efforts required to implement the BIST circuitry and the loss of the fault coverage for the use of pseudo-random patterns have prevented the BIST architecture from being widely used in industry. As the semiconductor manufacturing technology moves into the nanometer design era, it remains to be seen how the CBILBO-based architecture proposed in [Wang 1986b], which can always guarantee 100% single stuck-at fault coverage and has the ability of running 10 times more BIST patterns than the STUMPS-based architecture, will perform. Challenges lie ahead with regard to whether or not pseudo-exhaustive testing will become a preferred BIST pattern generation technique.

Because the primary objective of this chapter is to familiarize the reader with basic DFT techniques, many advanced DFT techniques, along with novel *design-for-reliability* (DFR), *design-for-manufacturability* (DFM), *design-for-yield* (DFY), *design-for-debug-and-diagnosis* (DFD), and low-power test techniques, are left out. For advanced reading, the reader is referred to [Gizopoulos 2006; Wang 2006a, 2007a]. These techniques are of growing importance to help us cope with the physical failures of the nanometer design era.

The DFT chapter is the first of a series of three chapters devoted to VLSI testing. These chapters are chosen to equip the reader with basic DFT skills to design quality digital circuits. Chapter 7 discusses the design rules and test synthesis steps required to implement testability logic into these digital circuits. Chapter 14 jumps into the important fault simulation and test generation techniques for generating quality test patterns to screen defective chips from manufacturing test.

## 3.7 EXERCISES

**3.1.** (**Testability Analysis**) Calculate the SCOAP controllability and observability measures for a 3-input XOR gate and for its NAND-NOR implementation.

**3.2.** (**Testability Analysis**) Use the rules given in Tables 3.3 and 3.4 to calculate the probability-based testability measures for a 3-input XNOR gate and for its NAND-NOR implementation. Assume that the

probability-based controllability values at all primary inputs and the probability-based observability value at the primary output are 0.5 and 1, respectively.

**3.3.** (**Testability Analysis**) Repeat Exercise 3.2 for the full-adder circuit shown in Figure 3.1.

**3.4.** (**Muxed-D Scan Cell**) Show a possible CMOS implementation of the muxed-D scan cell shown in Figure 3.5a.

**3.5.** (**Low-Power Muxed-D Scan Cell**) Design a low-power version of the muxed-D scan cell given in Figure 3.5a by adding gated-clock logic that includes a lock-up latch to control the clock port.

**3.6.** (**At-Speed Scan**) Assume that a scan design contains three clock domains running at 100 MHz, 200 MHz, and 400 MHz, respectively. In addition, assume that the clock skew between any two clock domains is manageable. List all possible at-speed scan ATPG methods and compare their advantages and disadvantages in terms of fault coverage and test pattern count.

**3.7.** (**At-Speed Scan**) Describe two major capture-clocking schemes for at-speed scan testing and compare their advantages and disadvantages. Also discuss what will happen if three or more captures are used.

**3.8.** (**BIST Pattern Generation**) Implement a period-8 in-circuit *test pattern generator* (TPG) with a binary counter. Compare its advantages and disadvantages with a Johnson counter (twisted-ring counter).

**3.9.** (**BIST Pattern Generation**) Implement a period-31 in-circuit *test pattern generator* (TPG) with a modular *linear feedback shift register* (LFSR) with *characteristic polynomial* $f(x) = 1 + x^2 + x^5$. Convert the modular LFSR into a muxed-D scan design with minimum area overhead.

**3.10.** (**BIST Pattern Generation**) Implement a period-31 in-circuit *test pattern generator* (TPG) with a five-stage *cellular automaton* (CA) with *construction rule = 11001, where* "0" denotes a *rule 90* cell and "1" denotes a *rule 150* cell. Convert the CA into an LSSD design with minimum area overhead.

**3.11.** (**Cellular Automata**) Derive a construction rule for a cellular automaton of length 54, and then construction rules up to length 300 to match the list of primitive polynomials up to degree 300 reported in [Bardell 1987].

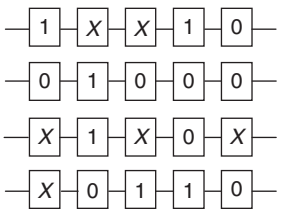
**3.12.** (**BIST Response Compaction**) Discuss in detail what errors can and cannot be detected by a MISR.

**3.13.** (**STUMPS *versus* CBILBO**) Compare the performance of a STUMPS design and a CBILBO design. Assume that both designs operate at 400 MHz and that the circuit under test has 100 scan chains each having 1000 scan cells. Compute the test time for each design when 100,000 test patterns are to be applied. In general, the shift (scan) speed is much slower than a circuit's operating speed. Assume that

the scan shift frequency is 50 MHz, and compute the test time for the STUMPS design again. Explain further why the STUMPS-based architecture is gaining more popularity than the CBILBO-based architecture.

**3.14.** (**Scan *versus* Logic BIST *versus* Test Compression**) Compare the advantages and disadvantages of a scan design, a logic BIST design, and a test compression design in terms of fault coverage, test application time, test data volume, and area overhead.

**3.15.** (**Test Stimulus Compression**) Given a circuit with four scan chains, each having five scan cells, and with a set of test cubes listed:




**a.** Design the multiple-input broadcast scan decompressor that fulfills the test cube requirements.

**b.** What is the compression ratio?

**c.** The assignment of *X*'s will affect the compression performance dramatically. Give one X-assignment example that will unfortunately lead to no compression with this multiple-input broadcast scan decompressor.

**3.16.** (**Test Stimulus Compression**) Derive mathematical expressions for the following in terms of the number of tester channels, $n$, and the expansion ratio, $k$.

**a.** The probability of encoding a scan slice containing 2 specified bits with Illinois scan.

**b.** The probability of encoding a scan slice containing 3 specified bits, where each scan chain is driven by the XOR of a unique combination of 2 tester channels such that there are a total of $C_2^n = n(n-1)/2$ scan chains.

**3.17.** (**Test Stimulus Compression**) For the sequential linear decompressor shown in Figure 3.38 whose corresponding system of linear equations is shown in Figure 3.39, find the compressed stimulus, $X_1 - X_{10}$, necessary to encode the following test cube: $< Z_1, \ldots, Z_{12} > = <0\text{---}1\text{-}0\text{--}010>$.

**3.18.** (**Test Stimulus Compression**) For the MUX network shown in Figure 3.43 and then the XOR network shown in Figure 3.44a, find the compressed stimulus at the network inputs necessary to encode the following test cube: <1 - 0 - - - 01>.

**3.19.** **(Test Response Compaction)** Explain further how many errors and how many unknowns (*X*'s) can be detected or tolerated by the *X*-compactor and *q*-compactor as shown in Figures 3.47 and 3.52, respectively.

**3.20.** **(Test Response Compaction)** For the X-compact matrix of the X-compactor given below:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

**a.** What is the compaction ratio?

**b.** Which outputs after compaction are affected by the second scan chain output?

**c.** How many errors can be detected by the X-compactor?

## ACKNOWLEDGMENTS

I wish to thank Dr. Xinghao Chen of CTC Technologies for contributing the Testability Analysis section; Professor Xiaowei Li and Professor Yinhe Han of Chinese Academy of Sciences, Professor Kuen-Jong Lee of National Cheng Kung University, Professor Nur A. Touba of the University of Texas at Austin for contributing a portion of the Circuits for Test Stimulus Compression and Circuits for Test Response Compaction sections. I also express my gratitude to Professor Xiaoqing Wen of Kyushu Institute of Technology, Professor Nur A. Touba of the University of Texas at Austin, Professor Kewal K. Saluja of the University of Wisconsin–Madison, Professor Subhasish Mitra of Stanford University, Dr. Rohit Kapur and Khader S. Abdel-Hafez of Synopsys, Dr. Brion Keller of Cadence Design Systems, and Dr. Benoit Nadeau-Dostie of LogicVision for reviewing the text and providing helpful comments, and Teresa Chang of SynTest Technologies for drawing most of the figures.

## REFERENCES

### R3.0 Books

[Abramovici 1994] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design,* IEEE Press, Revised Printing, Piscataway, NJ, 1994.

[Bardell 1987] P. H. Bardell, W. H. McAnney, and J. Savir, *Built-In Test for VLSI: Pseudorandom Techniques,* John Wiley & Sons, Somerset, NJ, 1987.

[Bushnell 2000] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing for Digital, Memory & Mixed-Signal VLSI Circuits,* Springer, Boston, 2000.

[Crouch 1999] A. Crouch, *Design for Test for Digital IC's and Embedded Core Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1999.

[Gizopoulos 2006] D. Gizopoulos, editor, *Advances in Electronic Testing: Challenges and Methodologies,* Morgan Kaufmann, San Francisco, 2006.

[Golomb 1982] S. W. Golomb, *Shift Register Sequence,* Aegean Park Press, Laguna Hills, CA, 1982.

[Jha 2003] N. Jha and S. Gupta, *Testing of Digital Systems,* Cambridge University Press, London, 2003.

[McCluskey 1986] E. J. McCluskey, *Logic Design Principles: With Emphasis on Testable Semicustom Circuits,* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Mourad 2000] S. Mourad and Y. Zorian, *Principles of Testing Electronic Systems,* John Wiley & Sons, Somerset, NJ, 2000.

[Nadeau-Dostie 2000] B. Nadeau-Dostie, *Design for At-Speed Test, Diagnosis and Measurement,* Springer, Boston, 2000.

[Peterson 1972] W. W. Peterson and E. J. Weldon, Jr., *Error-Correcting Codes,* MIT Press, Cambridge, MA, 1972.

[Rajski 1998] J. Rajski and J. Tyszer, *Arithmetic Built-In Self-Test for Embedded Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1998.

[Stroud 2002] C. E. Stroud, *A Designer's Guide to Built-In Self-Test,* Springer, Boston, 2002.

[Wang 2006a] L.-T. Wang, C.-W. Wu, and X. Wen, editors, *VLSI Test Principles and Architectures: Design for Testability,* Morgan Kaufmann, San Francisco, 2006.

[Wang 2007a] L.-T. Wang, C. E. Stroud, and N. A. Touba, editors, *System-on-Chip Test Architectures: Nanometer Design for Testability,* Morgan Kaufmann, San Francisco, 2007.

## R3.1 Introduction

[Fujiwara 1982] H. Fujiwara and S. Toida, The complexity of fault detection problems for combinational circuits, *IEEE Trans. on Computers*, C-31(6), pp. 555–560, June 1982.

[SIA 2005] SIA, *The International Technology Roadmap for Semiconductors: 2005 Edition—Design,* Semiconductor Industry Association, San Jose, CA, http://public.itrs.net, 2005.

[SIA 2006] SIA, *The International Technology Roadmap for Semiconductors: 2006 Update,* Semiconductor Industry Association, San Jose, CA, http://public.itrs.net, 2006.

[Touba 2006] N. A. Touba, Survey of test vector compression techniques, *IEEE Design & Test of Computers*, 23(4), pp. 294–303, July–August 2006.

## R3.2 Testability Analysis

[Agrawal 1982] V. D. Agrawal and M. R. Mercer, Testability measures—What do they tell us?, in *Proc. IEEE Int. Test Conf.*, pp. 391–396, November 1982.

[Breuer 1978] M. A. Breuer, New concepts in automated testing of digital circuits, in *Proc. EEC Symp. on CAD of Digital Electronic Circuits and Systems*, pp. 69–92, November 1978.

[Goldstein 1979] L. H. Goldstein, Controllability/Observability analysis of digital circuits, *IEEE Trans. on Circuits and Systems*, CAS-26(9), pp. 685–693, September 1979.

[Goldstein 1980] L. H. Goldstein and E. L. Thigpen, SCOAP: Sandia controllability/observability analysis program, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 190–196, June 1980.

[Grason 1979] J. Grason, TMEAS—a testability measurement program, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 156–161, June 1979.

[Ivanov 1988] A. Ivanov and V. K. Agarwal, Dynamic testability measures for ATPG, *IEEE Trans. on Computer-Aided Design*, 7(5), pp. 598–608, May 1988.

[Jain 1985] S. K. Jain and V. D. Agrawal, Statistical fault analysis, *IEEE Design & Test of Computers*, 2(2), pp. 38–44, February 1985.

[Parker 1975] K. P. Parker and E. J. McCluskey, Probability treatment of general combinational networks, *IEEE Trans. on Computers*, 24(6), pp. 668–670, June 1975.

[Rizzolo 2001] R. F. Rizzolo, B. F. Robbins, and D. G. Scott, A hierarchical approach to improving random pattern testability on IBM eServer z900 chips, in *Digest of Papers, IEEE North Atlantic Test Workshop*, pp. 84–89, May 2001.

[Rutman 1972] R. A. Rutman, Fault detection test generation for sequential logic heuristic tree search, *IEEE Computer Repository*, Paper R-72-187, September/October 1972.

[Savir 1984] J. Savir, G. S. Ditlow, and P. H. Bardell, random pattern testability, *IEEE Trans. on Computer*, C-33(1), pp. 79–90, January 1984.

[Seth 1985] S. C. Seth, L. Pan, and V. D. Agrawal, PREDICT—Probabilistic estimation of digital circuit testability, in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 220–225, June 1985.

[Stephenson 1976] J. E. Stephenson and J. Garson, A testability measure for register transfer level digital circuits, in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 101–107, June 1976.

[Wang 1984] L.-T. Wang and E. Law, Daisy testability analyzer (DTA), in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 143–145, November 1984.

[Wang 1985] L.-T. Wang and E. Law, An enhanced Daisy testability analyzer (DTA), in *Proc. Automatic Testing Conf.*, pp. 223–229, October 1985.

## R3.3 Scan Design

[Cheung 1997] B. Cheung and L.-T. Wang, The seven deadly sins of scan-based designs, in *Integrated System Design*, www.eetimes.com/editorial/1997/test9708.html, August 1997.

[DasGupta 1982] S. DasGupta, P. Goel, R. G. Walther, and T. W. Williams, A variation of LSSD and its implications on design and test pattern generation in VLSI, in *Proc. IEEE Int. Test Conf.*, pp. 63–66, November 1982.

[Eichelberger 1977] E. B. Eichelberger and T. W. Williams, A logic design structure for LSI testability, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 462–468, June 1977.

[Nadeau-Dostie 1994] B. Nadeau-Dostie, A. Hassan, D. Burek, and S. Sunter, Multiple Clock Rate Test Apparatus for Testing Digital Systems, U.S. Patent No. 5,349,587, September 20, 1994.

[Rajski 2003] J. Rajski, A. Hassan, R. Thompson, and N. Tamarapalli, Method and Apparatus for At-Speed Testing of Digital Circuits, U.S. Patent Application No. 20030097614, May 22, 2003.

[Savir 1993] J. Savir and S. Patil, Scan-based transition test, *IEEE Trans. on Computer-Aided Design*, 12(8), pp. 1232–1241, August 1993.

[Savir 1994] J. Savir and S. Patil, Broad-side delay test, *IEEE Trans. on Computer-Aided Design*, 13(8), pp. 1057–1064, August 1994.

[Wang 2005a] L.-T. Wang, M.-C. Lin, X. Wen, H.-P. Wang, C.-C. Hsu, S.-C. Kao, and F.-S. Hsu, Multiple-Capture DFT System for Scan-Based Integrated Circuits, U.S. Patent No. 6,954,887, October 11, 2005.

[Wang 2007b] L.-T. Wang, P.-C. Hsu, and X. Wen, Multiple-Capture DFT System for Detecting or Locating Crossing Clock-Domain Faults During Scan-Test, U.S. Patent No. 7,260,756, August 21, 2007.

## R3.4 Logic Built-In Self-Test

[Bardell 1982] P. H. Bardell and W. H. McAnney, Self-testing of multiple logic modules, in *Proc. IEEE Int. Test Conf.*, pp. 200–204, November 1982.

[Barzilai 1981] Z. Barzilai, J. Savir, G. Markowsky, and M. G. Smith, The weighted syndrome sums approach to VLSI testing, *IEEE Trans. on Computers*, 30(12), pp. 996–1000, December 1981.

[Barzilai 1983] Z. Barzilai, D. Coppersmith, and A. Rosenberg, Exhaustive bit pattern generation in discontiguous positions with applications to VLSI testing, *IEEE Trans. on Computers*, 32(2), pp. 190–194, February 1983.

[Benowitz 1975] N. Benowitz, D. F. Calhoun, G. E. Alderson, J. E. Bauer, and C. T. Joeckel, An advanced fault isolation system for digital logic, *IEEE Trans. on Computers*, 24(5), pp. 489–497, May 1975.

[Cadence 2008] Cadence Design Systems, http://www.cadence.com, 2008.

[Chen 1987] C. L. Chen, Exhaustive test pattern generation with cyclic codes, *IEEE Trans. on Computers*, 37(3), pp. 329–338, March 1987.

[Cheon 2005] B. Cheon, E. Lee, L.-T. Wang, X. Wen, P. Hsu, J. Cho, J. Park, H. Chao, and S. Wu, At-speed logic BIST for IP cores, in *Proc. IEEE/ACM Design, Automation, and Test in Europe Conf.*, pp. 860–861, March 2005.

[Chin 1984] C. K. Chin and E. J. McCluskey, *Weighted Pattern Generation for Built-In Self-Test*, Center for Reliable Computing, Technical Report (CRC TR) No. 84-7, Stanford University, August 1984.

[Foote 1997] T. G. Foote, D. E. Hoffman, W. V. Huott, T. J. Koprowski, B. J. Robbins, and M. P. Kusko, Testing the 400 MHz IBM generation-4 CMOS chip, in *Proc. IEEE Int. Test Conf.*, pp. 106–114, November 1997.

[Frohwerk 1977] R. A. Frohwerk, Signature analysis: A new digital field service method, in *Hewlett-Packard J.*, 28, pp. 2–8, September 1977.

[Furukawa 2006] H. Furukawa, X. Wen, L.-T. Wang, B. Sheu, Z. Jiang, and S. Wu, A novel and practical control scheme for inter-clock at-speed testing, in *Proc. IEEE Int. Test Conf.*, Paper 17.2, October 2006.

[Gloster 1988] C. S. Gloster, Jr. and F. Brglez, Boundary scan with cellular built-in self-test, in *Proc. IEEE Int. Test Conf.*, pp. 138–145, September 1988.

[Hassan 1984] S. Z. Hassan and E. J. McCluskey, Increased fault coverage through multiple signatures, in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 354–359, June 1984.

[Hayes 1976] J. P. Hayes, Transition count testing of combinational logic circuits, *IEEE Trans. on Computers*, C-25(6), pp. 613–620, June 1976.

[Hortensius 1989] P. D. Hortensius, R. D. McLeod, W. Pries, D. M. Miller, and H. C. Card, Cellular automata-based pseudorandom number generators for built-in self-test, *IEEE Trans. on Computer-Aided Design*, 8(8), pp. 842–859, August 1989.

[Keller 2007] B. Keller, A. Uzzaman, B. Li, and T. Snethen, Using programmable on-product clock generation (OPCG) for delay test, in *Proc. IEEE Asian Test Symp.*, pp. 69–72, October 2007.

[Khara 1987] M. Khara and A. Albicki, Cellular automata used for test pattern generation, in *Proc. IEEE Int. Conf. on Computer Design*, pp. 56–59, October 1987.

[Könemann 1979] B. Könemann, J. Mucha, and G. Zwiehoff, Built-in logic block observation techniques, in *Proc. IEEE Int. Test Conf.*, pp. 37–41, October 1979.

[Könemann 1980] B. Könemann, J. Mucha, and G. Zwiehoff, Built-in test for complex digital circuits, *IEEE J. of Solid-State Circuits*, 15(3), pp. 315–318, June 1980.

[Lai 2007] L. Lai, W.-T. Cheng, and T. Rinderknecht, Programmable scan-based logic built-in self test, in *Proc. IEEE Asian Test Symp.*, pp. 371–377, October 2007.

[LogicVision 2008] LogicVision, http://www.logicvision.com, 2008.

[McCluskey 1981] E. J. McCluskey and S. Bozorgui-Nesbat, Design for autonomous test, *IEEE Trans. on Computers*, 30(11), pp. 860–875, November 1981.

[McCluskey 1984] E. J. McCluskey, Verification testing—A pseudoexhaustive test technique, *IEEE Trans. on Computers*, 33(6), pp. 541–546, June 1984.

[McCluskey 1985] E. J. McCluskey, Built-in self-test structures, *IEEE Design & Test of Computers*, 2(2), pp. 29–36, April 1985.

[Mentor 2008] Mentor Graphics, http://www.mentor.com, 2008.

[Nadeau-Dostie 1994] B. Nadeau-Dostie, A. Hassan, D. Burek, and S. Sunter, Multiple Clock Rate Test Apparatus for Testing Digital Systems, U.S. Patent No. 5,349,587, September 20, 1994.

[Nadeau-Dostie 2006] B. Nadeau-Dostie and J.-F. Côté, Clock Controller for At-Speed Testing of Scan Circuits, U.S. Patent No. 7,155,651, December 26, 2006.

[Nadeau-Dostie 2007] B. Nadeau-Dostie, Method and Circuit for At-Speed Testing of Scan Circuits, U.S. Patent No. 7,194,669, March 20, 2007.

[Rajski 2003] J. Rajski, A. Hassan, R. Thompson, and N. Tamarapalli, Method and Apparatus for At-Speed Testing of Digital Circuits, U.S. Patent Application No. 20030097614, May 22, 2003.

[Savir 1980] J. Savir, Syndrome-testable design of combinational circuits, *IEEE Trans. on Computers*, 29(6), pp. 442–451, June 1980.

[Savir 1985] J. Savir and W. H. McAnney, On the masking probability with ones count and transition count, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 111–113, November 1985.

[Schnurmann 1975] H. D. Schnurmann, E. Lindbloom, and R. G. Carpenter, The weighted random test-pattern generator, *IEEE Trans. on Computers*, 24(7), pp. 695–700, July 1975.

[SynTest 2008] SynTest Technologies, http://www.syntest.com, 2008.

[Tang 1984] D. T. Tang and C. L. Chen, Logic test pattern generation using linear codes, *IEEE Trans. on Computers*, 33(9), pp. 845–850, September 1984.

[Tsai 1999] H.-C. Tsai, K.-T. Cheng, and S. Bhawmik, Improving the test quality for scan-based BIST using a general test application scheme, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 748–753, June 1999.

[van Sas 1990] J. van Sas, F. Catthoor, and H. D. Man, Cellular automata-based self-test for programmable data paths, in *Proc. IEEE Int. Test Conf.*, pp. 769–778, September 1990.

[Wang 1986a] L.-T. Wang and E. J. McCluskey, Condensed linear feedback shift register (LFSR) testing—A pseudoexhaustive test technique, *IEEE Trans. on Computers*, 35(4), pp. 367–370, April 1986.

[Wang 1986b] L.-T. Wang and E. J. McCluskey, Concurrent built-in logic block observer (CBILBO), in *Proc. IEEE Int. Symp. on Circuits and Systems*, 3(3), pp. 1054–1057, May 1986.

[Wang 1987] L.-T. Wang and E. J. McCluskey, Linear feedback shift register design using cyclic codes, *IEEE Trans. on Computers*, 37(10), pp. 1302–1306, October 1987.

[Wang 1988a] L.-T. Wang and E. J. McCluskey, Hybrid designs generating maximum-length sequences, *Special Issue on Testable and Maintainable Design, IEEE Trans. on Computer-Aided Design*, 7(1), pp. 91–99, January 1988.

[Wang 1988b] L.-T. Wang and E. J. McCluskey, Circuits for pseudo-exhaustive test pattern generation, *IEEE Trans. on Computer-Aided Design*, 7(10), pp. 1068–1080, October 1988.

[Wang 1989] L.-T. Wang, M. Marhoefer, and E. J. McCluskey, A self-test and self-diagnosis architecture for boards using boundary scan, in *Proc. IEEE European Test Conf.*, pp. 119–126, April 1989.

[Wang 2005b] L.-T. Wang, X. Wen, P.-C. Hsu, S. Wu, and J. Guo, At-speed logic BIST architecture for multi-clock designs, in *Proc. Int. Conf. on Computer Design*, pp. 475–478, October 2005.

[Wang 2006b] L.-T. Wang, P.-C. Hsu, S.-C. Kao, M.-C. Lin, H.-P. Wang, H.-J. Chao, and X. Wen, Multiple-Capture DFT System for Detecting or Locating Crossing Clock-Domain Faults During Self-Test or Scan-Test, U.S. Patent No. 7,007,213, February 28, 2006.

[Williams 1987] T. W. Williams, W. Daehn, M. Gruetzner, and C. W. Starke, Aliasing errors in signature analysis registers, *IEEE Design & Test of Computers*, 4(2), pp. 39–45, April 1987.

[Wolfram 1983] S. Wolfram, Statistical mechanics of cellular automata, in *Review of Modern Physics*, 55(3), pp. 601–644, July 1983.

[Wunderlich 1987] H.-J. Wunderlich, Self test using unequiprobable random patterns, in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 258–263, July 1987.

## R3.5 Test Compression

[Barnhart 2002] C. Barnhart, V. Brunkhorst, F. Distler, O. Farnsworth, A. Ferko, B. Keller, D. Scott, B. Koenemann, and T. Onodera, Extending OPMISR beyond 10x scan test efficiency, *IEEE Design & Test of Computers*, 19(5), pp. 65–73, May-June 2002.

[Bayraktaroglu 2001] I. Bayraktaroglu and A. Orailoglu, Test volume and application time reduction through scan chain concealment, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 151–155, June 2001.

[Bayraktaroglu 2003] I. Bayraktaroglu and A. Orailoglu, Concurrent application of compaction and compression for test time and data volume reduction in scan designs, *IEEE Trans. on Computers*, 52(11), pp. 1480–1489, November 2003.

[Beck 2005] M. Beck, O. Barondeau, M. Kaibel, F. Poehl, X. Lin, and R. Press, Logic design for on-chip test clock generation—Implementation details and impact on delay test quality, in *Proc. IEEE/ACM Design, Automation, and Test in Europe Conf.*, pp. 56–61, March 2005.

[Cadence 2008] Cadence Design Systems, http://www.cadence.com, 2008.

[Chakrabarty 1998] K. Chakrabarty, B. T. Murray, and J. P. Hayes, Optimal zero-aliasing space compaction of test responses, *IEEE Trans. on Computers*, 47(11), pp. 1171–1187, November 1998.

[Chandra 2007] A. Chandra, H. Yan, and R. Kapur, Multimode Illinois scan architecture for test application time and test data volume reduction, in *Proc. IEEE VLSI Test Symp.*, pp. 84–92, May 2007.

[Fan 2007] X.-X. Fan, Y. Hu, and L.-T. Wang, An on-chip test clock control scheme for multi-clock at-speed testing, in *Proc. IEEE Asian Test Symp.*, pp. 341–348, October 2007.

[Frohwerk 1977] R. A. Frohwerk, Signature analysis: A new digital field service method, in *Hewlett-Packard J.*, 28, pp. 2–8, September 1977.

[Furukawa 2006] H. Furukawa, X. Wen, L.-T. Wang, B. Sheu, Z. Jiang, and S. Wu, A novel and practical control scheme for inter-clock at-speed testing, in *Proc. IEEE Int. Test Conf.*, Paper 17.2, October 2006.

[Gu 2001] X. Gu, S. S. Chung, F. Tsang, J. A. Tofte, and H. Rahmanian, An effort-minimized logic BIST implementation method, in *Proc. IEEE Int. Test Conf.*, pp. 1002–1010, October 2001.

[Hamzaoglu 1999] I. Hamzaoglu and J. H. Patel, Reducing test application time for full scan embedded cores, in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 260–267, July 1999.

[Han 2006] Y. Han, X. Li, H. Li, and A. Chandra, Embedded test resource for SoC to reduce required tester channels based on advanced convolutional codes, *IEEE Trans. on Instrumentation and Measurement*, 55(2), pp. 389–399, April 2006.

[Han 2007] Y. Han, Y. Hu, X. Li, H. Li, and A. Chandra, Embedded test decompressor to reduce the required channels and vector memory of tester for complex processor circuit, *IEEE Trans. on Very Large Scale Integration Systems*, 15(5), pp. 531–540, May 2007.

[Hsu 2001] F. F. Hsu, K. M. Butler, and J. H. Patel, A case study on the implementation of Illinois scan architecture, in *Proc. IEEE Int. Test Conf.*, pp. 538–547, October 2001.

[Kapur 2008] R. Kapur, S. Mitra, and T. W. Williams, Historical perspective on scan compression, *IEEE Design & Test of Computers*, 25(2), pp. 114–120, March-April 2008.

[Keller 2007] B. Keller, A. Uzzaman, B. Li, and T. Snethen, Using programmable on-product clock generation (OPCG) for delay test, in *Proc. IEEE Asian Test Symp.*, pp. 69–72, October 2007.

[Könemann 1991] B. Koenemann, LFSR-coded test patterns for scan designs, in *Proc. IEEE European Test Conf.*, pp. 237–242, April 1991.

[Könemann 2001] B. Koenemann, C. Barnhart, B. Keller, T. Snethen, O. Farnsworth, and D. Wheater, A SmartBIST variant with guaranteed encoding, in *Proc. IEEE Asian Test Symp.*, pp. 325–330, November 2001.

[Könemann 2003] B. Koenemann, C. Barnhart, and B. Keller, Real-Time Decoder for Scan Test Patterns, U.S. Patent No. 6,611,933, August 26, 2003.

[Krishna 2001] C. V. Krishna, A. Jas, and N. A. Touba, Test vector encoding using partial LFSR reseeding, in *Proc. IEEE Int. Test Conf.*, pp. 885–893, October 2001.

[Krishna 2003] C. V. Krishna and N. A. Touba, Adjustable width linear combinational scan vector decompression, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 863–866, September 2003.

[Lee 1998] K.-J. Lee, J. J. Chen, and C. H. Huang, Using a single input to support multiple scan chains, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 74–78, November 1998.

[Lee 1999] K.-J. Lee, J. J. Chen, and C. H. Huang, Broadcasting test patterns to multiple circuits, *IEEE Trans. on Computer-Aided Design*, 18(12), pp. 1793–1802, December 1999.

[Li 2004] L. Li and K. Chakrabarty, Test set embedding for deterministic BIST using a reconfigurable interconnection network, *IEEE Trans. on Computer-Aided Design*, 23(9), pp. 1289–1305, September 2004.

[LogicVision 2008] LogicVision, http://www.logicvision.com, 2008.

[Mentor 2008] Mentor Graphics, http://www.mentor.com, 2008.

[Mitra 2004a] S. Mitra and K. S. Kim, X-Compact: An efficient response compaction technique, *IEEE Trans. on Computer-Aided Design*, 23(3), pp. 421–432, March 2004.

[Mitra 2004b] S. Mitra, S. S. Lumetta, and M. Mitzenmacher, X-tolerant signature analysis, in *Proc. IEEE Int. Test Conf.*, pp. 432–441, October 2004.

[Mitra 2006] S. Mitra and K. S. Kim, XPAND: An efficient test stimulus compression technique, *IEEE Trans. on Computers*, 55(2), pp. 163–173, February 2006.

[Mrugalski 2004] G. Mrugalski, J. Rajski, and J. Tyszer, Ring generators—new devices for embedded test applications, *IEEE Trans. on Computer-Aided Design*, 23(9), pp. 1306–1320, September 2004.

[Nadeau-Dostie 2004] B. Nadeau-Dostie, Method of Masking Corrupt Bits During Signature Analysis and Circuit for Use Therewith, U.S. Patent No. 6,745,359, June 1, 2004.

[Nadeau-Dostie 2005] B. Nadeau-Dostie, J.-F. Côté, and F. Maamari, Structural test with functional characteristics, in *Proc. IEEE Current and Defect-Based Testing Workshop*, pp. 57–60, May 2005.

[Nadeau-Dostie 2006] B. Nadeau-Dostie and J.-F. Côté, Clock Controller for At-Speed Testing of Scan Circuits U.S. Patent No. 7,155,651, December 26 2006.

[Naruse 2003] M. Naruse, I. Pomeranz, S. M. Reddy, and S. Kundu, On-chip compression of output responses with unknown values using LFSR reseeding, in *Proc. IEEE Int. Test Conf.*, pp. 1060–1068, October 2003.

[Pandey 2002] A. R. Pandey and J. H. Patel, Reconfiguration technique for reducing test time and test volume in Illinois scan architecture based designs, in *Proc. IEEE VLSI Test Symp.*, pp. 9–15, April 2002.

[Patel 2003] J. H. Patel, S. S. Lumetta, and S. M. Reddy, Application of Saluja-Karpovsky compactors to test responses with many unknowns, in *Proc. IEEE VLSI Test Symp.*, pp. 107–112, April 2003.

[Pouya 1998] B. Pouya and N. A. Touba, Synthesis of zero-aliasing space elementary-tree space compactors, in *Proc. IEEE VLSI Test Symp.*, pp. 70–77, April 1998.

[Rajski 2004] J. Rajski, J. Tyszer, M. Kassab, and N. Mukherjee, Embedded deterministic test, *IEEE Trans. on Computer-Aided Design*, 23(5), pp. 776–792, May 2004.

[Rajski 2005] J. Rajski, J. Tyszer, C. Wang, and S. M. Reddy, Finite memory test response compactors for embedded test applications, *IEEE Trans. on Computer-Aided Design*, 24(4), pp. 622–634, April 2005.

[Rajski 2008] J. Rajski, J. Tyszer, G. Mrugalski, W.-T. Cheng, N. Mukherjee, and M. Kassab, X-Press: Two-stage X-tolerant compactor with programmable selector, *IEEE Trans. on Computer-Aided Design*, 27(1), pp. 147–159, January 2008.

[Saluja 1983] K. K. Saluja and M. Karpovsky, Test compression hardware through data compression in space and time, in *Proc. IEEE Int. Test Conf.*, pp. 83–88, October 1983.

[Samaranayake 2003] S. Samaranayake, E. Gizdarski, N. Sitchinava, F. Neuveux, R. Kapur, and T. W. Williams, A reconfigurable shared scan-in architecture, in *Proc. IEEE VLSI Test Symp.*, pp. 9–14, April 2003.

[Shah 2004] M. A. Shah and J. H. Patel, Enhancement of the Illinois scan architecture for use with multiple scan inputs, in *Proc. IEEE Computer Society Annual Symp. on VLSI*, pp. 167–172, February 2004.

[Sitchinava 2004] N. Sitchinava, S. Samaranayake, R. Kapur, E. Gizdarski, F. Neuveux, and T. W. Williams, Changing the scan enable during shift, in *Proc. IEEE VLSI Test Symp.*, pp. 73–78, April 2004.

[Synopsys 2008] Synopsys, http://www.synopsys.com, 2008.

[SynTest 2008] SynTest Technologies, http://www.syntest.com, 2008.

[Touba 2006] N. A. Touba, Survey of test vector compression techniques, *IEEE Design & Test of Computers*, 23(4), pp. 294–303, July-August 2006.

[Touba 2007] N. A. Touba, X-canceling MISR—An X-tolerant methodology for compacting output responses with unknowns using a MISR, in *Proc. IEEE Int. Test Conf.*, Paper 6.2, October 2007.

[Touba 2008] N. A. Touba and L.-T. Wang, X-Canceling Multiple-Input Signature Register (MISR) for Compacting Output Responses with Unknowns, U.S. Patent Application No. 12,007,693, January 14, 2008.

[Volkerink 2005] E. H. Volkerink and S. Mitra, Response compaction with any number of unknowns using a new LFSR architecture, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 117–122, June 2005.

[Wang 2002] L.-T. Wang, H.-P. Wang, X. Wen, M.-C. Lin, S.-H. Lin, D.-C. Yeh, S.-W. Tsai, K. S. Abdel-Hafez, Method and Apparatus for Broadcasting Scan Patterns in a Scan-Based Integrated Circuit, U.S. Patent Application No. 20030154433, January 16, 2002.

[Wang 2004] L.-T. Wang, X. Wen, H. Furukawa, F.-S. Hsu, S.-H. Lin, S.-W. Tsai, K. S. Abdel-Hafez, and S. Wu, VirtualScan: A new compressed scan technology for test cost reduction, in *Proc. IEEE Int. Test Conf.*, pp. 916–925, October 2004.

[Wang 2008] L.-T. Wang, X. Wen, S. Wu, Z. Wang, Z. Jiang, B. Sheu, and X. Gu, VirtualScan: Test compression technology using combinational logic and one-pass ATPG, *IEEE Design & Test of Computers*, 25(2), pp. 122–130, March-April 2008.

[Wohl 2001] P. Wohl, J. A. Waicukauski, and T. W. Williams, Design of compactors for signature-analyzers in built-in self-test, in *Proc. IEEE Int. Test Conf.*, pp. 54–63, October 2001.

[Wohl 2003] P. Wohl, J. A. Waicukauski, S. Patel, and M. B. Amin, Efficient compression and application of deterministic patterns in a logic BIST architecture, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 566–569, June 2003.

[Wohl 2004] P. Wohl, J. A. Waicukauski, and S. Patel, Scalable selector architecture for X-tolerant deterministic BIST, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 934–939, June 2004.

[Wohl 2007a] P. Wohl, J. A. Waicukauski, R. Kapur, S. Ramnath, E. Gizdarski, T. W. Williams, and P. Jaini, Minimizing the impact of scan compression, in *Proc. IEEE VLSI Test Symp.*, pp. 67–74, May 2007.

[Wohl 2007b] P. Wohl, J. A. Waicukauski, and S. Ramnath, Fully X-tolerant combinational scan compression, in *Proc. IEEE Int. Test Conf.*, Paper 6.1, October 2007.

## R3.6 Concluding Remarks

[Bardell 1982] P. H. Bardell and W. H. McAnney, Self-testing of multiple logic modules, in *Proc. IEEE Int. Test Conf.*, pp. 200–204, November 1982.

[Naffziger 2006] S. Naffziger, B. Stackhouse, T. Grutkowski, D. Josephson, J. Desai, E. Alon, and M. Horowitz, The implementation of a 2-core multi-threaded Itanium family processor, *IEEE J. of Solid-State Circuits*, 41(1), pp. 197–209, January 2006.

[Stackhouse 2008] B. Stackhouse, B. Cherkauer, M. Gowan, P. Gronowski, and C. Lyles, A 65 nm 2-billion-transistor quad-core Itanium processor, *Digest of Papers, IEEE Int. Solid-State Circuits Conf.*, pp. 92, February 2008.

[Wang 1986b] L.-T. Wang and E. J. McCluskey, Concurrent built-in logic block observer (CBILBO), in *Proc. IEEE Int. Symp. on Circuits and Systems*, 3(3), pp. 1054–1057, May 1986.

CHAPTER

# 14 Fault simulation and test generation

James C.-M. Li
*National Taiwan University, Taipei, Taiwan*

Michael S. Hsiao
*Virginia Tech, Blacksburg, Virginia*

## ABOUT THIS CHAPTER

***Very large-scale integration*** (VLSI) circuits can be defective because of the imperfect manufacturing process. One of the most important tasks in VLSI testing is to minimize the number of defective chips shipped to customers. The quality of test patterns is critical in determining the thoroughness of testing. This requires the assessment of the quality of test patterns either developed manually or generated automatically so that a desired product quality can be achieved.

This chapter consists of two major VLSI testing topics: fault simulation and test generation. In fault simulation, we start with a discussion on fault collapsing. After an introduction of equivalent faults and dominant faults, the serial, parallel, concurrent, and differential fault simulation techniques are described, followed by qualitative comparisons between their advantages and drawbacks. These techniques trade accuracy for reduced execution time, which is crucial for managing the complexity of large designs. After fault simulation, basic ***automatic test pattern generation*** (ATPG) techniques, including Boolean difference, PODEM, and FAN, are described in detail. Advanced test generation techniques are also introduced to meet the demand for quality testing, including sequential ATPG, delay fault ATPG, and bridging fault ATPG. Throughout this chapter, the reader will learn about the major fault simulation and test generation techniques. This background will be valuable in selecting the test method that best meets the design needs and understands the relationship between test patterns and product quality.

## 14.1 INTRODUCTION

Simulation techniques have been widely used in VLSI designs for digital circuit verification, test development, design debug, and fault diagnosis. During the design stage, **logic simulation**, which has been extensively discussed in

Chapter 8, is performed to help verify whether the design meets its specifications and contains any design errors. It also helps locate design errors that may have escaped from detection during design debug.

Once the design meets its specifications and is ready for physical implementation, one must ensure that the manufactured devices will function as intended and no defective parts are shipped to customers. To achieve high product quality, typically with a defect level less than 500 ***defective parts per million*** (DPM), quality test patterns must be developed. At present, as we move to the nanometer design era, this has required applying **fault simulation** and ***automatic test pattern generation*** (ATPG) to the design that has been embedded with ***design for testability*** (DFT) features during test development.

In contrast to logic simulation, **fault simulation** is used to measure the effectiveness of test patterns in detecting defects that might have been introduced during the manufacturing process. This requires simulating the faulty behavior of the circuit in detecting the modeled faults of interest. (For this reason, logic simulation is generally referred to as **fault-free simulation**.) Furthermore, fault simulation is an integral component of any ATPG program.

The major difference between logic simulation and fault simulation lies in the nature of the non-idealities they deal with. Logic simulation is intended for checking whether the circuit's responses to a given set of input vectors conform to the given specifications or a known good design as the reference. Design errors may be introduced by human designers or ***electronic design automation*** (EDA) tools, and they should be caught before physical implementation. **Fault simulation**, on the other hand, is concerned with checking the behavior of fabricated circuits as a consequence of inevitable fabrication process imperfections. The manufacturing defects (*e.g.*, wire shorts and opens), if present, may cause the circuits to behave differently from the expected behavior. Fault simulation generally assumes that the design is functionally correct (*i.e.*, free of design errors), and it is targeted at capturing manufacturing defects. However, we note that fault simulation methods may be applied during the design verification stage as well.

The capability of fault simulation to predict the faulty circuit behavior is of great importance for test and diagnosis. First, fault simulation evaluates the effectiveness of a set of test patterns in detecting manufacturing defects. The quality of a test set is expressed in terms of **fault coverage**, which is the ratio of detected faults to the total number of faults in the circuit. In practice, the designer uses a **fault simulator** to evaluate the fault coverage of a set of input stimuli (test vectors or test patterns) with respect to the modeled faults of interest. Because fault simulation concerns the fault coverage of a test set rather than the detection of design bugs, it is also termed **fault grading**. Low fault coverage test patterns will jeopardize the manufacturing test quality and eventually lead to unacceptable field returns from customers. Second, fault simulation helps to identify undetected faults, which is especially important when the achieved fault coverage is below an acceptable level. In this case, either the designer or

the ATPG has to generate additional test vectors to improve the fault coverage (*i.e.*, to detect those remaining undetected faults). Third, as part of the **test compaction** process, fault simulation identifies redundant test patterns, which may be discarded with no negative impact on the fault coverage. With the preceding capabilities and applications, fault simulation is one of the crucial components of ATPG. In fact, the implementation of an ATPG program usually starts with the fault simulator. Finally, fault simulation assists **fault diagnosis**, which determines the type and location of faults that best explain the faulty circuit behavior of the device under diagnosis. The fault simulation results are compared with the observed circuit responses to identify the most likely faults. The fault type and location information can then be used as a starting point for locating the defects that cause the circuit malfunction.

Although logic and fault simulators can provide important information about the behavior of the circuit, they require a set of test vectors with which the circuit is simulated. The objective of test generation, then, is the task of producing a set of test vectors that will uncover any defect in a chip. Figure 14.1 illustrates a high-level concept of test generation. In this figure, the circuit at the top is defect free, and for any defective chip that is functionally different from the defect-free one there must exist some input that can differentiate the two. Generating effective test patterns efficiently for a digital circuit is thus the goal of an ATPG system.

Because this problem is extremely difficult, DFT methods have been frequently used to relieve the burden on the ATPG. In this sense, a powerful ATPG can be regarded as the "Holy Grail" in testing, with which all DFT methods could potentially be eliminated. In other words, if the ATPG engine is capable of efficiently delivering high-quality test patterns that achieve high fault coverages and small test sets on large, complex chips, DFT would no longer be necessary.

Because fault simulation can help to determine those faults that could be detected by the same generated test, it becomes an essential component of ATPG. By removing those incidentally detected faults, ATPG is able to significantly

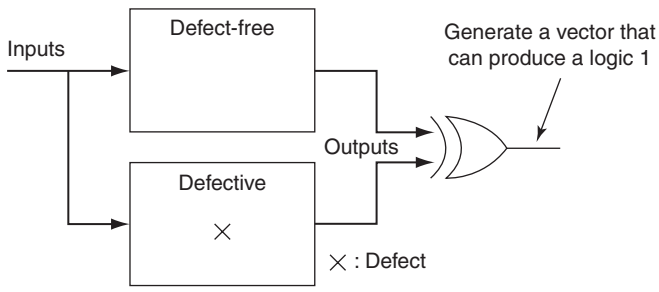



**FIGURE 14.1**

Conceptual view of test generation.

reduce the number of faults that it needs to consider after the generation of each new test vector, thereby improving the efficiency of the ATPG process.

For some fault models, the circuit layout information is needed. For example, wire delay values are needed to compute the longest paths, and the actual positions of gates and wires are needed to identify those likely bridges. However, because ATPG is a time-consuming process, we would like to start the ATPG process before the layout is available. In this regard, an ATPG may be performed to obtain an initial test set without the layout information. Then, after **place and route**, any faults that require circuit layout information that are undetected by the test set would be identified, and the ATPG can be invoked again to target these specific undetected faults to ensure test quality.

## 14.2 FAULT COLLAPSING

**Fault collapsing** reduces the number of faults to be considered in fault simulation and ATPG so the overall run time can be reduced. Two requirements must be met for fault collapsing to become effective. First, fault collapsing must run much faster than fault simulation or ATPG; otherwise, fault collapsing may not be worth doing. Second, the collapsed faults must be representative of all original faults modeled in the circuit. In this section, we introduce two fault-collapsing techniques: **equivalence fault collapsing** and **dominance fault collapsing**. Linear time algorithms are given to meet the first requirement. We illustrate that dominance fault collapsing produces a fewer number of faults than equivalence fault collapsing. However, from a fault coverage accuracy viewpoint, equivalence fault collapsing is more often quoted than dominance fault collapsing, because the former results in a better indication of the test quality.

### 14.2.1 Equivalence fault collapsing

Let two faults $f$ and $g$ be said to be **functionally equivalent** (or simply **equivalent**) if the faulty outputs of these two faults are identical for any input [McCluskey 1971; Abramovici 1994; Bushnell 2000]. Equivalent faults are **indistinguishable**, because there is no test pattern that can tell them apart. Consider the example of a two-input AND gate shown in Figure 14.2. The good outputs and faulty outputs of the AND gate for all four possible input combinations are listed in Table 14.1. From this table, we can see that $A$ stuck-at zero fault (denoted as $A/0$) and $C$ stuck-at zero fault (denoted as $C/0$) are equivalent.

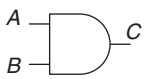



**FIGURE 14.2**

An example two-input AND gate.

**Table 14.1** Good and faulty outputs for Figure 14.2

| Input | | Output | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *A* | *B* | *C* | *A*/0 | *C*/0 | *B*/0 | *A*/1 | *C*/1 | *B*/1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | **1** | **1** | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | **1** |
| 1 | 1 | 1 | **0** | **0** | **0** | 1 | 1 | 1 |

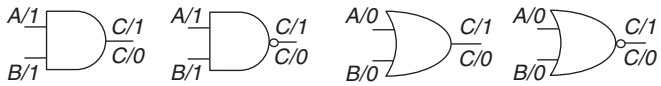



**FIGURE 14.3**

Equivalence collapsed fault list for four elementary gates.

This is because the faulty outputs of these two faults are always the same for all the four input combinations. On the other hand, the $A$ stuck-at one fault ($A/1$) and the $C$ stuck-at one fault ($C/1$) are not equivalent, because the input pattern $A = B = 0$ can distinguish these two faults. Another input pattern is $A = 1$ and $B = 0$. For clear illustration, the faulty outputs that are different from good outputs are underlined and highlighted in bold.

The equivalence relationship is **symmetric**. This means, if fault $f$ is equivalent to fault $g$, then fault $g$ is equivalent to fault $f$. The equivalence relationship is also **transitive**. That is, if fault $f$ is equivalent to fault $g$ and fault $g$ is equivalent to fault $h$, then fault $f$ is equivalent to fault $h$. For the example given in Figure 14.2, $A/0$ fault is equivalent to $B/0$ fault, and $B/0$ fault is equivalent to $C/0$ fault. These three faults $\{A/0, B/0, C/0\}$ belong to the same **equivalence class**.

**Equivalence fault collapsing** reduces the set of faults that needs to be considered with the fault equivalence relation. Only one representative fault is selected from every equivalent class. Figure 14.3 shows the **equivalence collapsed fault list** for four types of elementary gates. Originally, there are six faults associated with a two-input AND gate: $A/1$, $A/0$, $B/0$, $B/1$, $C/0$, and $C/1$. After equivalence fault collapsing, the number of faults is reduced to four: $A/1$, $B/1$, $C/1$, and $C/0$. The other types of gates can be examined in the same way. Generally speaking, an $n$-input elementary gate has $2n$ and $n + 2$ stuck-at faults before and after equivalence fault collapsing, respectively. Note that the equivalence fault collapsed fault list is not unique, and there are other ways to collapse the faults than are shown in Figure 14.3. For example, $\{A/0, A/1, B/1, C/1\}$ is another possible way to perform equivalence fault collapsing.

Equivalence fault collapsing can be performed by either functional analysis or structural analysis. Exhaustive functional analysis is time-consuming, because enumeration of $2^n$ patterns may be needed for an $n$-input circuit (like Table 14.1 for the AND gate shown in Figure 14.2). Therefore, in the following text, we only demonstrate a linear-time structural analysis to perform equivalence fault collapsing. The resulting equivalence collapsed fault list may not be minimal, but structural analysis is good enough for most applications.

For a fanout-free circuit consisting of elementary gates (such as buffers, inverters AND, OR, NAND, and NOR gates), equivalence fault collapsing can be performed by keeping two kinds of faults: (1) both stuck-at one and stuck-at zero faults on every primary output, and (2) one collapsed fault on each gate input whose stuck value is shown in Figure 14.3. Inverters and buffers should be treated as wires. For the example in Figure 14.4, we keep both $H/0$ and $H/1$ faults on primary output $H$. We also keep one fault on each gate input, such as $A/0$ and $B/0$ for OR gate $G_1$, etc. Note that faults on the gate outputs are removed, because they are equivalent to some other faults in the figure. For example, gate $G_1$ output stuck-at zero fault is equivalent to $C/0$ fault, which is again equivalent to $E/1$ fault, which is in turn equivalent to $H/0$ fault.

For circuits with fanouts, fault collapsing becomes complicated, because faults on the fanout stem are now always equivalent to the faults on the fanout branches. Figure 14.5 shows a circuit with a fanout stem $E$ and two fanout branches $L$ and $F$. According to Table 14.2, $E/0$ fault is equivalent to $F/0$ fault but not equivalent to $L/0$ fault. Also, none of the stuck-at one faults are equivalent. **Stem analysis** is required to determine equivalent faults on a fanout stem and its branches. However, stem analysis is generally not cost-effective in terms of CPU time, so the details are skipped in this chapter.

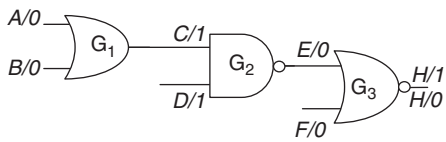



**FIGURE 14.4**

Equivalence fault collapsing on a fanout-free circuit.

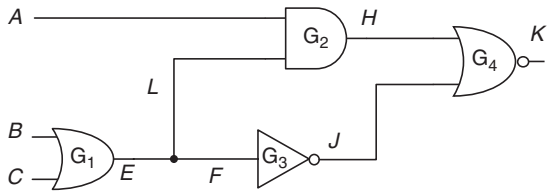



**FIGURE 14.5**

Equivalence fault collapsing for faults on fanouts.

**Table 14.2** Good and faulty outputs for Figure 14.5

| Input | | | Output | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *A* | *B* | *C* | *E* | *E*/0 | *F*/0 | *L*/0 | *E*/1 | *F*/1 | *L*/1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

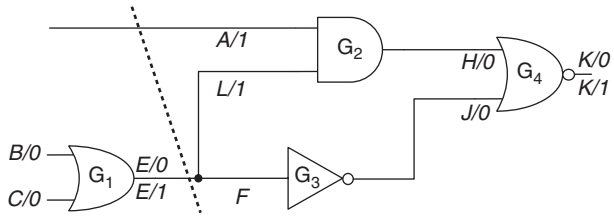



**FIGURE 14.6**

Equivalence collapsed fault list for Figure 14.5.

To avoid stem analysis, an approximation solution is used to partition the circuit into independent ***fanout-free regions*** (FFRs). Every fanout stem is treated as a primary output, so both stuck-at one and stuck-at zero faults are included in the collapsed fault list. Algorithm 14.1 introduces a simple equivalence fault-collapsing (simple_EFC) algorithm without stem analysis. Figure 14.6 shows the resulting equivalence collapsed fault list with the simple_EFC algorithm. The circuit is partitioned into two independent fanout-free regions: four faults in one region and six faults in the other. The simple_EFC algorithm reduces the number of faults from 18 to 10. Please note that inverter $G_3$ is ignored in this algorithm, because its input stuck-at one fault is always equivalent to its output stuck-at zero fault and *vice versa*. Also note that simple_EFC is not the only way to perform fault collapsing; other implementations of fault collapsing are possible.

**Algorithm 14.1** A simple equivalence fault-collapsing algorithm

```
simple_EFC (N) /*N is a netlist*/
 1. fault_list = {};
 2. foreach gate or PO or PI g in N
 3.   if ((g is PO) || (g is PI and fanout stem)) then
 4.     fault_list = fault_list ∪ g stuck-at 0 and 1;
 5.   else if (output of gate g is fanout stem) then
 6.     fault_list = fault_list ∪ g output stuck-at 0 and 1;
 7.   end if
 8.   if (gate g is AND) || (gate g is NAND) then
 9.     fault_list = fault_list ∪ g input stuck-at 1;
10.   else if (gate g is OR) || (gate g is NOR) then
11.     fault_list = fault_list ∪ g input stuck-at 0;
12.   end if
13. end foreach
14. return (fault_list);
```

The simple_EFC algorithm can complete in linear time because it checks every gate exactly once. However, this algorithm has two drawbacks. First, the result is not optimal, because it lacks stem analysis. For example, Table 14.2 shows that *E*/0 fault is actually equivalent to *K*/0 fault, but they both appear in Figure 14.6. This small error, however, is often acceptable in most cases. Second, the relationship between the original (uncollapsed) faults and the corresponding collapsed faults is lost. For example, the link is lost between the four faults {*F*/0, *J*/1, *H*/1, *K*/0} in the same **equivalence class** and their collapsed fault *K*/0. The relation between **uncollapsed faults** and **collapsed faults** is needed when calculating the fault coverage of the circuit. Fault coverage can be calculated on the basis of either the uncollapsed faults or the collapsed faults. The **uncollapsed fault coverage** is the number of detected uncollapsed faults over the total number of uncollapsed faults, whereas the **collapsed fault coverage** is the number of detected collapsed faults over the total number of collapsed faults. Oftentimes, these two numbers are not identical but close to each other. Missing the link between the collapsed faults and the uncollapsed fault makes it difficult to convert the collapsed fault coverage to the uncollapsed fault coverage. However, modern fault simulators and ATPG programs have found an easy way to rebuild the link by performing another pass of linear-time analysis on equivalent faults.

### 14.2.2 Dominance fault collapsing

The equivalence collapsed fault list can be further compressed with the **fault dominance** relationship. Let the **detecting set** of fault $f$ (denoted as $T_f$) be the set of all test patterns that detect fault $f$. Fault $f$ dominates fault $g$ if the

detecting set of fault $f$ contains that of fault $g$. That means, $T_f \supseteq T_g$. For the example in Figure 14.2, fault $C/1$ dominates fault $A/1$ because the detecting set of $C/1$ {00, 01, 10} contains the detecting set of $A/1$ {01}. The dominance relation is not symmetric but is transitive.

If a test pattern detects the **dominated fault**, then it must detect the corresponding **dominating fault**. To reduce the run time, the dominating faults can be removed from the fault list. The reduction of fault list with the fault dominance relation is called **dominance fault collapsing**. If two faults are equivalent, then they dominate each other. Therefore, the number of dominance-collapsed faults must be smaller or equal to that of equivalence-collapsed faults.

Figure 14.7 shows the **dominance collapsed fault list** of four elementary gates. Originally, there are four equivalence-collapsed faults for a two-input AND gate: $A/1$, $B/1$, $C/0$, and $C/1$. After dominance fault collapsing, the number of faults is reduced to three: $A/1$, $B/1$, and $C/0$. The other types of gates can be examined in the same way. Generally speaking, for an n-input elementary gate, there are $n + 1$ stuck-at faults after dominance fault collapsing.

For a fanout-free circuit consisting of elementary gates (such as buffers, inverters AND, OR, NAND, and NOR gates), dominance fault collapsing can be performed according to the following two rules: (1) one collapsed fault on every primary input whose value is shown in Figure 14.7, and (2) one collapsed fault on each gate output whose gate inputs are all primary inputs. Those gates whose inputs are all primary inputs are called **input gates**. Inverters and buffers should be treated as wires. Figure 14.8 shows the dominance collapsed fault list of the example fanout-free circuit. Note that no fault is needed on $G_2$ gate output, because $G_2$ is not an input gate. $E/0$ fault dominates $C/1$ fault, so the former can be removed. *E/1* fault is equivalent to *C/0* fault, which dominates *A/0* fault, so both *C/0* and *E/1* faults are removed. The explanation of the other faults is similar so it is left as an exercise for the readers. This circuit has 14 uncollapsed faults, which are reduced to 8 equivalent faults and then to 5 dominant faults after equivalence and dominance fault collapsing, respectively.

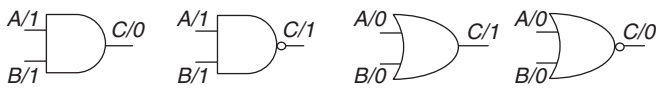



**FIGURE 14.7**

Dominance collapsed fault list for four elementary gates.

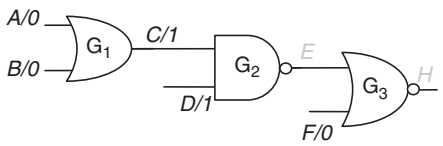



**FIGURE 14.8**

Dominance fault collapsing on a fanout-free circuit.

Faults on the fanout branches do not always dominate faults on the fanout stem. Consider again the example in Figure 14.5. According to Table 14.2, *F*/1 fault dominates *E*/1 fault. However, *L*/1 fault does not dominate *E*/1 fault. (Actually, *L*/1 fault has an empty detecting set so *L*/1 is a **redundant fault**. More details on redundant faults are given in the test generation section.) Again, stem analysis is needed to determine whether fanout branch faults dominate fanout stem faults.

An approximation method to avoid stem analysis is to partition the circuit into fanout-free regions and perform fault collapsing on each fanout-free region independently. A simple_DFC algorithm is shown in Algorithm 14.2. The dominance fault collapsed result is shown in Figure 14.9. The number of faults is reduced to seven. Without stem analysis, the result is not optimal because *J*/0 is equivalent to *F*/1, which dominates *E*/1.

**Algorithm 14.2** A simple dominance fault-collapsing algorithm

```
simple_DFC (N) /*N is a netlist*/
 1. fault_list = {};
 2. foreach gate or PI or PO g in N
 3.   if ((g is PI and fanout stem) || (g is PO and fanout branch)) then
 4.     fault_list = fault_list ∪ g output stuck-at 0 and 1;
 5.   else if (g is gate) then
 6.     foreach gate input i of gate g
 7.         h = backtrace inverters starting from i;
 8.         if (h is PI or fanout branch) then /* rule #1 */
 9.           if (gate g is AND) || (gate g is NAND) then
10.             fault_list = fault_list ∪ i stuck-at 1;
11.           else if (gate g is OR) || (gate g is NOR)
12.             fault_list = fault_list ∪ i stuck-at 0;
13.           end if
14.         end if
15.     end foreach
16.     if (every input of g has a fault) then /* rule #2 */
17.         if (gate g is AND) || (gate g is NOR) then
18.           fault_list = fault_list ∪ g output stuck-at 0;
19.         else if (gate g is OR) || (gate g is NAND) then
20.           fault_list = fault_list ∪ g output stuck-at 1;
21.         end if
22.     end if
23.   end if
24. end foreach
25. return (fault_list);
```

Although the dominance collapsed fault list is smaller than the equivalence collapsed fault list, fault coverage of the former is not as representative as that of the latter. The reason is that a test pattern may detect a dominating fault without

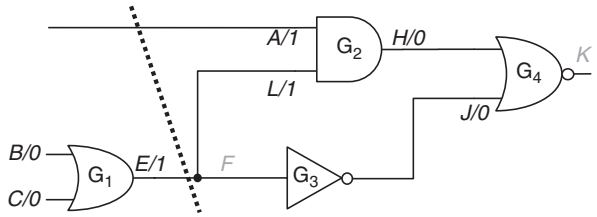



**FIGURE 14.9**

Dominance collapsed fault list for Figure 14.5.

detecting the dominated fault. For the example given in Figure 14.5, test pattern ABC = 100 does not detect the dominated fault *E*/1 but it detects the dominating fault *F*/1. If the dominance collapsed fault list is used during fault simulation, the dominance collapsed fault coverage may underestimate the test quality. As a result, modern fault simulators and ATPG programs favor the use of equivalence fault collapsing only.

## 14.3 FAULT SIMULATION

Fault simulation is a more challenging task than logic simulation because of the added dimension of complexity (*i.e.*, the behavior of the circuit containing all the modeled faults must be simulated). When simulating one fault at a time, the amount of computation is approximately proportional to the circuit size, the number of test patterns, and the number of modeled faults. Because the number of modeled faults are roughly proportional to the circuit size, the overall time complexity of fault simulation is $O(pn^2)$, for $p$ test patterns and $n$ logic gates, which becomes infeasible for large circuits. To improve fault simulation performance, various fault simulation techniques have been developed. In this section, we restrict our discussion to the single stuck-at fault model and illustrate the key fault simulation techniques along with qualitative comparisons between their advantages and drawbacks.

### 14.3.1 Serial fault simulation

Serial fault simulation is the simplest fault simulation technique. It consists of fault-free and faulty circuit simulations. Initially, fault-free logic simulation is performed on the original circuit to obtain the fault-free output responses. The fault-free responses are stored and later used to determine whether a test pattern can detect a fault or not. After fault-free simulation, a serial fault simulator simulates faults one at a time. For each fault, **fault injection** is first performed,

which modifies the original circuit to mimic the circuit behavior in the presence of the fault. Then, the faulty circuit is simulated to derive the *faulty* responses of the current fault with respect to the given test patterns. This process repeats until all faults in the fault list have been simulated.

The serial fault simulation process is demonstrated with the example circuit N. In this example, the fault list comprises two faults, $A$ stuck-at one (denoted by $f$) and $J$ stuck-at zero (denoted by $g$), which are depicted in Figure 14.10. Note that, although both faults are drawn in the figure, only one fault is present at a time under the single stuck-at fault model. The test set consists of three test patterns (denoted by $P_1$, $P_2$, $P_3$, respectively, and shown in the "Input" columns of Table 14.3).

The serial fault simulator starts from fault-free simulation. The fault-free responses are $K = \{1, 1, 0\}$ for input patterns $P_1$, $P_2$, and $P_3$, respectively. After the fault-free responses are available, fault $f$ is processed—fault injection is achieved by forcing $A$ to a constant one, and the obtained faulty circuit is simulated. The circuit responses for fault $f$ are $K_f = \{0, 0, 0\}$ with respect to the three input patterns. Compared with the fault-free responses (the "Output" column in Table 14.3), it is observed that patterns $P_1$ and $P_2$ detect fault $f$, but pattern $P_3$ does not. After fault $f$ has been simulated, circuit $N$ is restored by removing fault $f$. The next fault $g$ is then injected by forcing $J$ to zero. Simulation of the resulting faulty circuit is then performed to obtain the faulty outputs $K_g = \{1, 1, 1\}$ (also listed in Table 14.3). Fault $g$ is detected by pattern $P_3$ but not $P_1$ and $P_2$.

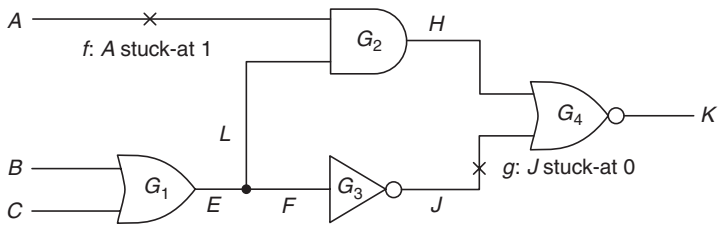



**FIGURE 14.10**

An example circuit with two faults.

**Table 14.3** Serial Fault Simulation Results for Figure 14.10

| Pat. # | Input | | | Internal | | | | | Output | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | $A$ | $B$ | $C$ | $E$ | $F$ | $L$ | $J$ | $H$ | $K_{good}$ | $K_f$ | $K_g$ |
| $P_1$ | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| $P_2$ | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| $P_3$ | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |

In this example, nine simulation runs are performed: three fault-free and six faulty circuit simulations. These nine simulation runs can be divided into three **simulation passes**. In each simulation pass, either the fault-free or the faulty circuit is simulated for the whole test pattern set. Thus, the first simulation pass consists of fault-free simulations for $P_1$, $P_2$, and $P_3$, and the second and third passes correspond to the faulty circuit simulations of faults $f$ and $g$, respectively, for $P_1$, $P_2$, and $P_3$.

By careful inspection of the simulation results in Table 14.3, one can observe that if we are only concerned with the set of faults that are detected by the test set $\{P_1, P_2, P_3\}$, simulations of the faulty circuit with fault $f$ for patterns $P_2$ and $P_3$ are redundant, because $f$ is already detected by $P_1$. (It is assumed that the test patterns are simulated in the order $P_1$, $P_2$, and then $P_3$.) Halting simulation of detected faults is called **fault dropping**. For the purpose of fault grading, fault dropping dramatically improves fault simulation performance, because most faults are detected after relatively few test patterns have been applied. Fault dropping, however, should be avoided in fault diagnosis applications in which the entire fault simulation results are usually required to facilitate the identification of the fault type and location.

The simplified serial fault simulation flow is depicted in Figure 14.11. Before fault simulation, *fault collapsing* is executed to reduce the size of the fault list, denoted by $F$. Fault-free simulation is then performed for all test patterns to obtain the correct responses $O_g$. The algorithm then proceeds to fault simulation. For each fault $f$ in $F$, if there exists a test pattern whose output response $O_f$ differs from that of the corresponding good circuit $O_g$, $f$ is removed from $F$, indicating that it is detected. When all patterns have been simulated, the remaining faults in $F$ are the undetected faults.

The major advantage of serial fault simulation is its ease of implementation—a regular logic simulator plus fault injection and output comparison procedures will suffice. In addition, serial fault simulation can handle a wide range of fault models, as long as the fault effects can be properly injected into the circuit. The major disadvantage of serial fault simulation is its low performance. As will be discussed in the following subsections, practical fault simulation techniques exploit parallelism and/or similarities among the faulty circuits to speed up the fault simulation process.

### 14.3.2 Parallel fault simulation

Similar to parallel logic simulation, fault simulation can take advantage of the bitwise parallelism inherent in the host computer to reduce fault simulation time. For instance, in a 32-bit wide CPU, logic operations (AND, OR, or XOR) can be performed on all 32 bits at once. There are two ways to realize bitwise parallelism in fault simulation: parallelism in faults and parallelism in patterns. These two approaches are referred to as **parallel fault simulation** and **parallel pattern fault simulation**.

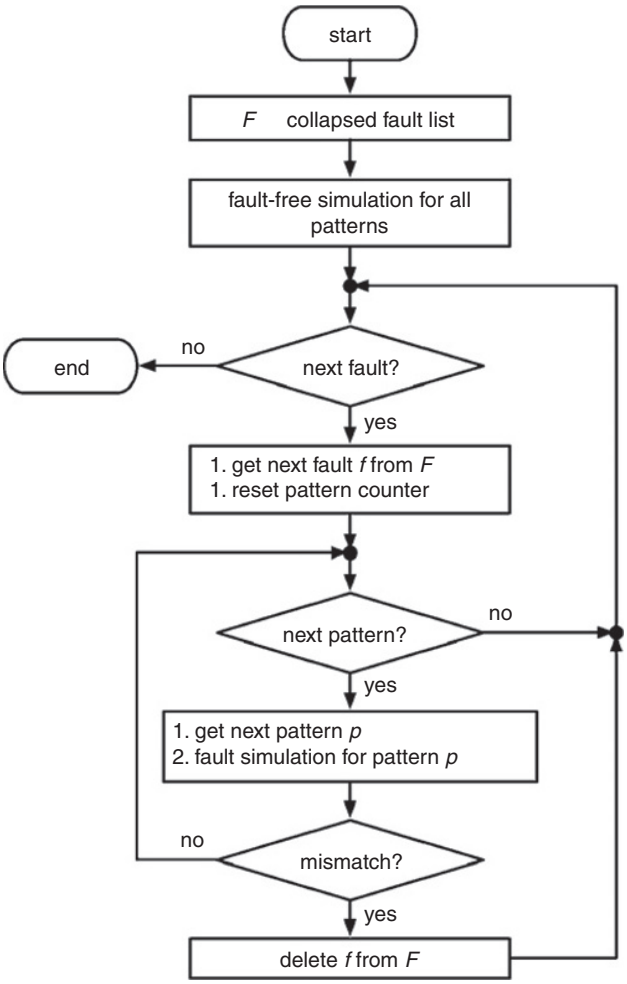



**FIGURE 14.11**

The serial fault simulation algorithm flow.

#### 14.3.2.1 *Parallel fault simulation*

Parallel fault simulation was proposed in the early 1960s [Seshu 1965]. Assuming that binary logic is used, one bit is sufficient to store the logic value of a signal. Thus, in a host computer that uses $w$-bit wide data words, each signal is associated with a data word of which $w$-1 bits are allocated for $w$-1 faulty circuits, and the remaining bit is reserved for the fault-free circuit. This way, $w$-1 faulty circuits and one fault-free circuit can be processed in parallel by use of bitwise logic operations, which corresponds to a speedup factor of approximately

$w$-1 compared with serial fault simulation. A fault is detected if its bit value differs from that of the fault-free circuit at any of the outputs.

We will reuse the example from serial fault simulation to illustrate the parallel fault simulation process. Assuming that the width of a computer word is three bits, the first bit stores the fault-free (FF) circuit response, and the second and third bits store the faulty responses in the presence of faults $f$ and $g$, respectively. The simulation results are shown in Table 14.4. Because the fault $f$, $A$ stuck-at 1, uses the second bit, it is injected by forcing the second bit of the data word of signal $A$ to 1 during fault simulation (shown in the "$A_f$" column with the forced value underlined—the "$A$" column corresponds to the fault-free case). Similarly, the "$J_g$" column depicts how fault $g$ is injected by forcing the third bit to 0.

As we have mentioned, parallel fault simulation is performed by use of bitwise logic operations. For example, the logic value of signal $H$ is obtained by a bitwise AND operation on the data words of signals $A$ and $L$. ($A$, $J$, and $L$ are circled in Table 14.4.) The faulty response of the first pattern is {1, **0**, 1}. This means that fault $f$ is detected (the second bit), but fault $g$ (the third bit) is not. Similarly, the outputs of $P_2$ and $P_3$ are {1 **0** 1} and {0 0 **1**}, respectively. In this example, three simulations (in one simulation pass) are performed. Compared with serial fault simulation, which requires nine simulations, parallel fault simulation saves two thirds of the simulation time.

To perform parallel fault simulation with regular parallel logic simulators, one may inject the faults by adding extra logic gates. Figure 14.12 shows how this is done for faults $f$ and $g$ in $N$. To inject $f$, a stuck-at one fault, an OR gate ($G_f$) is inserted, and to force the second bit of $A_f$ to be one without affecting the other two bits, the side input of $G_f$ is set to be 010. Note that the injection of fault $f$ does not affect the fault-free circuit and the faulty circuit with fault $g$.

**Table 14.4** Parallel fault simulation for Figure 14.10

| Pat # | | Input | | | | Internal | | | | | Output | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *A* | *Af* | *B* | *C* | *E* | *F* | *L* | *J* | *Jg* | *H* | *K* |
| $P_1$ | FF | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| | f | **0** | **1** | 1 | 0 | 1 | 1 | 1 | 0 | 0 | **1** | **0** |
| | g | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| $P_2$ | FF | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| | f | 0 | **1** | 0 | 1 | 1 | 1 | 1 | 0 | 0 | **1** | **0** |
| | g | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| $P_3$ | FF | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| | f | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| | g | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | **0** | 0 | **1** |

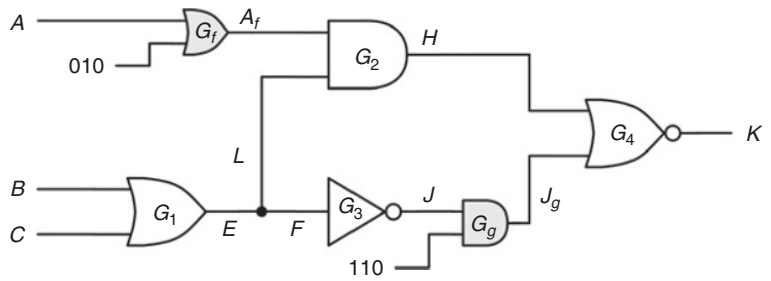



**FIGURE 14.12**

Fault injection for parallel fault simulation.

Similarly, injecting fault $g$, a stuck-at 0 fault, is achieved by adding the AND gate $G_g$ and setting its side input to be 110.

Note that the parallel fault simulation technique is applicable to the unit or zero delay models only. More complicated delay models cannot be modeled, because several faults are evaluated at the same time. Furthermore, a simulation pass cannot terminate unless all the faults in this pass are detected. For instance, we cannot drop fault $f$ alone after simulating pattern $P_1$, because fault $g$ is not detected yet. Parallel fault simulation is best used for simulating the beginning of the test pattern sequence, when a large number of faults are detected by each pattern.

### 14.3.2.2 *Parallel pattern fault simulation*

Bitwise parallelism can be used to simulate test patterns in parallel. For a host computer with a $w$-bit data width, the signal values for a sequence of $w$ test patterns are packed into a data word. For the fault-free or faulty circuit, $w$ test patterns can be simulated in parallel by use of bitwise logic operations. This approach was first reported in [Waicukauski 1985], in which it is called ***parallel pattern single fault propagation*** (PPSFP), because one fault at a time is simulated. This approach is especially useful for combinational circuits or full-scan sequential circuits.

In PPSFP, logic simulations on the fault-free circuit are first performed on the first $w$ test patterns, and the circuit outputs are recorded. Then, the faults are simulated one at a time on these $w$ test patterns. For each fault, the simulation results are compared with the correct responses to determine whether the fault is detected. Simulation continues until the fault is detected and dropped, or all the test patterns are simulated. The faulty circuit is restored to its original state, and the next fault is processed. The same procedure repeats until all faults in the fault list are simulated.

The PPSFP results of the fault simulation example are shown in Table 14.5. The "Fault-Free" row lists the fault-free simulation results. Note that the three patterns are packed into one single word and thus are evaluated simultaneously

**Table 14.5** PPSFP for Figure 14.10

| | | Input | | | Internal | | | | | Output |
|---|---|---|---|---|---|---|---|---|---|---|
| | | *A* | *B* | *C* | *E* | *F* | *L* | *J* | *H* | *K* |
| Fault Free | $P_1$ | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| | $P_2$ | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| | $P_3$ | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| f | $P_1$ | **1** | 1 | 0 | 1 | 1 | 1 | 0 | **1** | **0** |
| | $P_2$ | **1** | 0 | 1 | 1 | 1 | 1 | 0 | **1** | **0** |
| | $P_3$ | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| g | $P_1$ | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| | $P_2$ | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| | $P_3$ | 1 | 0 | 0 | 0 | 0 | 0 | **0** | 0 | **1** |

by use of bitwise logic operations. The "*f*" row represents the simulation results with fault *f* injected. In PPSFP, faults are injected by activating rising or falling events, depending on the stuck-at value, at the faulty signal. Thus, fault *f*, *A* stuck-at one, is injected by activating two rising events on input *A*. The faulty responses are {**0**, **0**, 0}, which indicates that fault *f* is detected by the first and second patterns but not the third one. After fault *f* is simulated, fault *f* is removed by activating two falling events on input *A* at pattern $P_1$ and $P_2$. Then, fault *g* is injected by activating one falling event on signal *J* at pattern $P_3$. A total of three simulation runs are carried out.

Figure 14.13 illustrates the simplified PPSFP flow. Again, fault collapsing is first executed to obtain the collapsed fault list *F*. Then, the first *w* patterns are simulated on the fault-free circuit in parallel, and the good outputs ($O_g$) are stored. Then, each fault *f* in the fault list *F* is simulated one by one with the same *w* test patterns. A fault is dropped and not simulated against the remaining test patterns if its output response $O_f$ is different from $O_g$. To fault simulate the next fault, the fault effect of the current fault is removed, and the next fault is injected. This process continues until all faults are either detected or simulated against all test patterns. If the number of test patterns is not an even multiple of the machine word width, only part of the machine word is used when simulating this last batch of patterns.

PPSFP is best suited for simulation of test patterns that come later in the test sequence, where the fault drop rate per pattern is lower. Parallel fault simulation does not work well in this situation, because it cannot terminate a simulation pass until all *w*-1 faults being processed are detected. PPSFP is not suitable for sequential circuits, because the circuit state for test pattern *i* in the *w*-bit

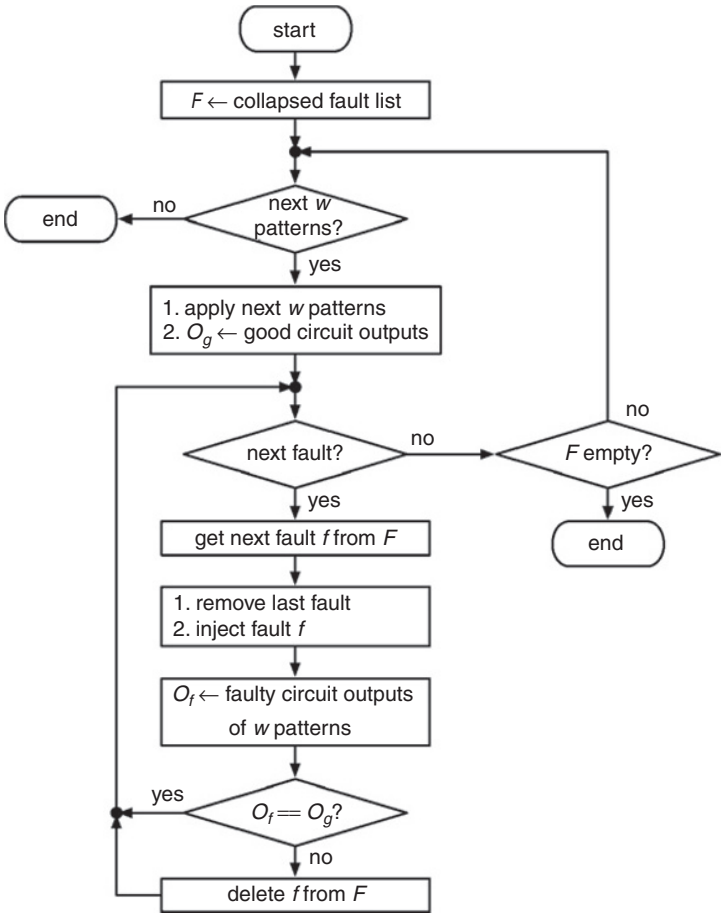



**FIGURE 14.13**

The PPSFP flowchart.

word depends on the previous $i$-1 patterns in the word, and this state is not available when the patterns are processed in parallel.

### 14.3.3 Concurrent fault simulation

Because a fault only affects the logic in the fanout cone from the fault site, the good circuit and faulty circuits typically only differ in a small region. **Concurrent fault simulation** exploits this fact and simulates only the differential parts of the whole circuit [Ulrich 1974]. Concurrent fault simulation is essentially an event-driven simulation with the fault-free circuit and faulty circuits simulated altogether.

In concurrent fault simulation, every gate has a **concurrent fault list**, which consists of a set of **bad gates**. A bad gate of gate $x$ represents an

imaginary copy of gate $x$ in the presence of a fault. Every bad gate contains a fault index and the associated gate I/O values in the presence of the corresponding fault. Initially, the concurrent fault list of gate $x$ contains **local faults** of gate $x$. The local faults of gate $x$ are faults on the inputs or outputs of gate $x$. As the simulation proceeds, the concurrent fault list contains not only local faults but also faults propagated from previous stages (called **fault effects**). Local faults of gate $x$ remain in the concurrent fault list of gate $x$ until they are detected.

Figure 14.14 illustrates the concurrent simulation of the example circuit for test pattern $P_1$. For clear illustration, we demonstrate three faults in this example: $A$ stuck-at one, $C$ stuck-at zero, and $J$ stuck-at zero faults. The concurrent fault lists with bad gates in grey are drawn beside the good gates. The fault indices are labeled in the middle of bad gates and their associated bad gate I/O values are labeled beside their I/O pins. The fault list of $G_1$, $G_2$, and $G_3$ initially contains their local faults: $C/0$, $A/1$, and $J/0$. When we apply the first pattern, three events occur in the primary inputs: u $\rightarrow$ 0 on A, u $\rightarrow$ 1 on B, and u $\rightarrow$ 0 on C. They are **good events**, because they happen in the good circuit. The output of good gate $G_1$ changes from unknown to one. In the presence of fault $C/0$, the output of faulty $G_1$ is the same as that of good $G_1$. A bad gate is **invisible** if its faulty output is the same as the good output. The bad gates C/0 and J/0 are both invisible so they are not propagated to the subsequent stages.

The output of $G_2$ changes from unknown to zero. In the presence of fault $A/1$, the faulty output changes from unknown to one. Because the faulty output differs from the good output, bad gate $A/1$ becomes visible. A bad gate is **visible** if its faulty output is different from the good output. The visible bad gate

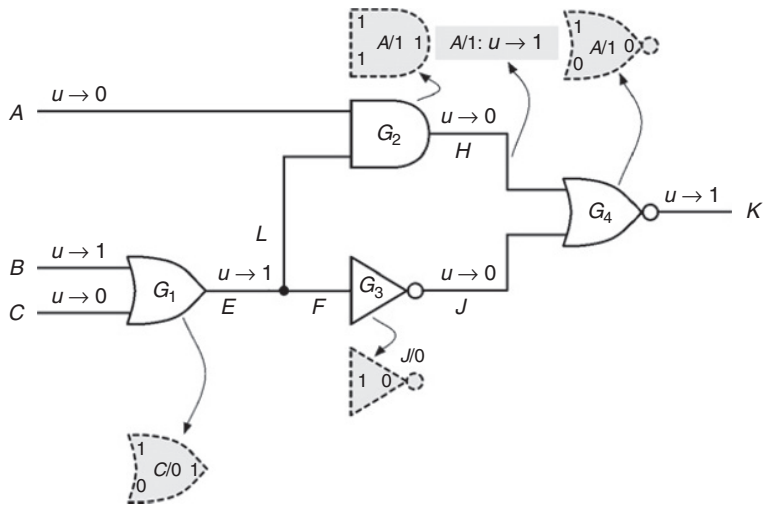



**FIGURE 14.14**

Concurrent fault simulation ($P_1$).

$A/1$ creates a bad event u → 1 on net $H$ (in gray). A **bad event** does not occur in the good circuit; it only occurs in the faulty circuit of the corresponding fault. A new copy of bad gate $A/1$ is added to the concurrent fault list of $G_4$, because it has one input different from the good gate. It is said that bad gate $A/1$ **diverges from** its good gate. Finally, fault $A/1$ is detected because the faulty output $K$ is different from the good output. At this time, we could drop detected fault $A/1$, but we keep it for illustration purposes.

Figure 14.15 illustrates the concurrent fault simulation for test pattern $P_2$. Two good events occur in this figure: $0 \rightarrow 1$ on C and $1 \rightarrow 0$ on $B$. The bad gate $C/0$, which was invisible in pattern $P_1$, now becomes **newly visible**. The newly visible bad gate creates a bad event, net $E$ falls to zero, which in turn creates two divergences in $G_2$ and $G_3$. The former is invisible, but the latter creates a bad event, net $J$ rises to one. Finally, the concurrent fault list of $G_4$ contains two bad gates; both faults $A/1$ and $C/0$ are detected.

For the last test pattern $P_3$ (Figure 14.16), two good events occur at primary inputs $A$ and $C$. The bad gate $C/0$ now becomes invisible. The bad gate $C/0$ is deleted from the concurrent fault list of $G_3$. A bad gate **converges to** its good gate if it is not a local fault and its I/O values are identical to those of the good gate. Similarly, the other bad gates $C/0$ also converge to $G_2$ and $G_4$. Note that bad gate $C/0$ does not converge to $G_1$, because it is a local fault for $G_1$. The bad gate $A/1$ can be examined in the same way. For gate $G_3$, although the faulty output of bad gate $J/0$ does not change, the good event $0 \rightarrow 1$ on $J$ makes bad gate $J/0$ newly visible.

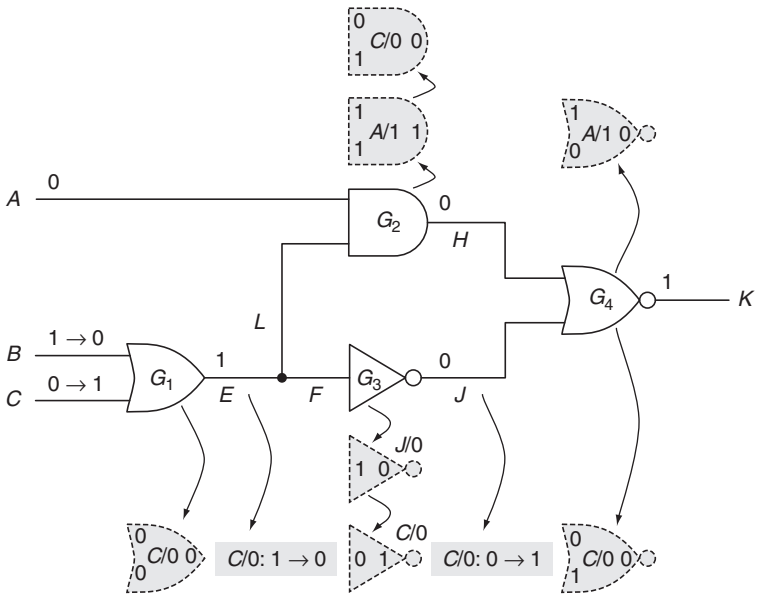



**FIGURE 14.15**

Concurrent fault simulation ($P_2$).

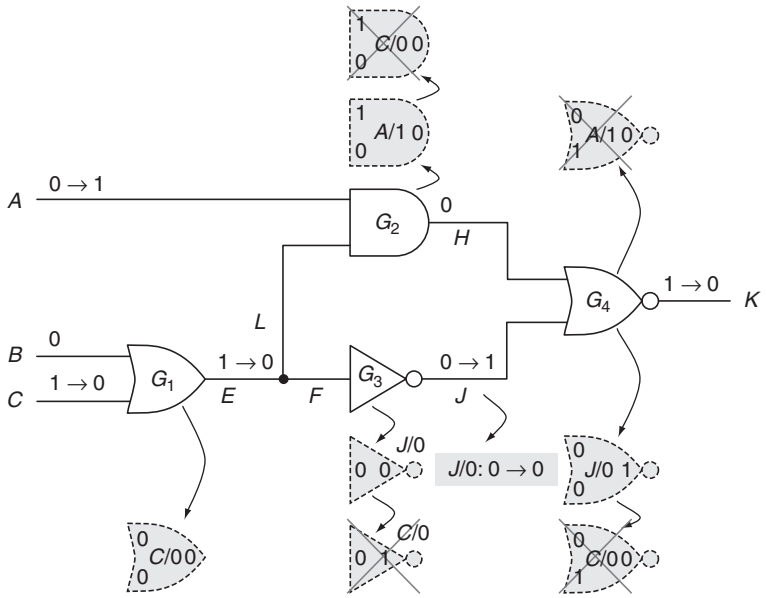



**FIGURE 14.16**

Concurrent fault simulation ($P_3$).

The newly visible event (in gray) is propagated to $G_4$, and a new bad gate $J/0$ diverges from $G_4$. Eventually, the fault $J/0$ is detected by pattern $P_3$.

Figure 14.17 shows a simplified concurrent fault simulation flowchart. The fault simulator applies one pattern at a time. The concurrent fault simulation is an event-driven simulation with both good events and bad events simulated at the same time. The events on the gate inputs are first analyzed. A good event affects both good and bad gates but a bad event only affects bad gates of the corresponding fault. After the analysis, events are then executed. The diverged bad gates and converged bad gates are added to or deleted from the fault list, respectively. Determining whether a bad gate diverges or converges depends on three factors: the visibility, the bad event, and the concurrent fault list (see [Abramovici 1994] for more details). After the event execution, new events are computed at the gate outputs. If an event reaches the primary outputs, detected faults can be removed from concurrent fault lists of all gates. This process repeats until there are no more test patterns, or no undetected faults.

### 14.3.4 **Differential fault simulation**

Concurrent fault simulation constructs the state of the faulty circuit from that of the same faulty circuit of the previous test pattern. Concurrent fault simulation has a potential memory problem, because the size of the concurrent fault list changes at runtime. In contrast, the single fault propagation technique

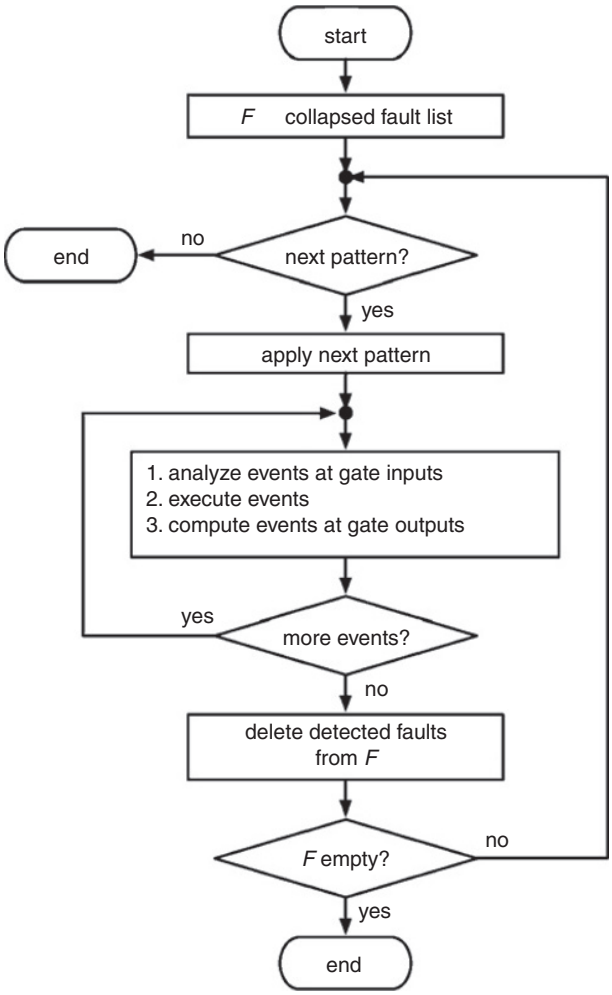
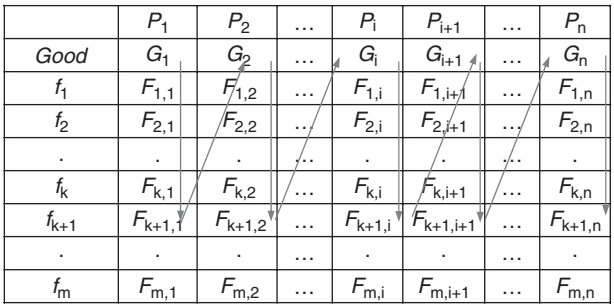



**FIGURE 14.17**

Concurrent fault simulation flowchart.

constructs the state of the faulty circuit from that of the good circuit. For sequential circuits, the single fault propagation technique would require a large overhead to store the states of the good circuit. Neither of the preceding two techniques are good for sequential fault simulation. Differential fault simulation combines the merits of concurrent fault simulation and single fault propagation techniques [Cheng 1989]. The idea is to simulate, in turn, every faulty circuit by tracking only the difference between a faulty circuit and the last simulated one. An event-driven simulator can easily implement differential fault simulation with the differences injected as events. This differential fault simulation technique



| | $P_1$ | $P_2$ | … | $P_i$ | $P_{i+1}$ | … | $P_n$ |
|---|---|---|---|---|---|---|---|
| *Good* | $G_1$ | $G_2$ | … | $G_i$ | $G_{i+1}$ | … | $G_n$ |
| $f_1$ | $F_{1,1}$ | $F_{1,2}$ | … | $F_{1,i}$ | $F_{1,i+1}$ | … | $F_{1,n}$ |
| $f_2$ | $F_{2,1}$ | $F_{2,2}$ | … | $F_{2,i}$ | $F_{2,i+1}$ | … | $F_{2,n}$ |
| . | . | . | … | . | . | … | . |
| $f_k$ | $F_{k,1}$ | $F_{k,2}$ | … | $F_{k,i}$ | $F_{k,i+1}$ | … | $F_{k,n}$ |
| $f_{k+1}$ | $F_{k+1,1}$ | $F_{k+1,2}$ | … | $F_{k+1,i}$ | $F_{k+1,i+1}$ | … | $F_{k+1,n}$ |
| . | . | . | … | . | . | … | . |
| $f_m$ | $F_{m,1}$ | $F_{m,2}$ | … | $F_{m,i}$ | $F_{m,i+1}$ | … | $F_{m,n}$ |

**FIGURE 14.18**

Differential fault simulation.

has been further combined with the parallel fault simulation technique, as implemented in **PROOFS** [Niermann 1992].

Figure 14.18 illustrates how differential fault simulation works. First, the first pattern $P_1$ is simulated on the good circuit $G_1$, and the good primary outputs are stored. Then the faulty circuit ($F_{1,1}$) is simulated with fault $f_1$ injected as an event. The first subscript indicates the fault and the second subscript indicates the pattern. The difference of states between $G_1$ and $F_{1,1}$ is stored. Note that only the states of storage elements, such as flip-flops, are stored, so the memory needed is small compared with concurrent fault simulation. If the primary outputs of $F_{1,1}$ and $G_1$ are not the same, then fault $f_1$ is detected. Following $F_1$ the second faulty circuit ($F_{2,1}$) is simulated with $f_1$ removed and $f_2$ injected. Similarly, the difference of states between $F_1$ and $F_2$ is stored. The preceding process continues until pattern $P_1$ has been simulated for all faults ($f_1$ to $f_m$).

Following the first pattern, the state of the good circuit $G_2$ is restored and the second pattern $P_2$ is applied. After the fault-free simulation, the primary outputs of $G_2$ are stored. The state of faulty circuit $F_{1,2}$ is restored by injecting the difference of $G_1$ and $F_{1,1}$. The fault $f_1$ is again injected as an event. The differential fault simulation for $P_2$ is the same as that of pattern $P_1$. Differential fault simulation goes in the direction of the arrows in Figure 14.18—$G_i$, $F_{1,i}$, $F_{2,i}$, . . . , $F_{m,i}$, $G_{i+1}$, $F_{1,i+1}$, . . . .

Figure 14.19 shows a simplified flowchart for differential fault simulation. For every test pattern, a fault-free simulation is performed first. Then the faulty circuits are simulated one after another. The states of every circuit are restored from the last simulation. If the faulty circuit outputs are different from the good outputs, the fault is detected and dropped. The state difference of every circuit is stored. With fault dropping, the state difference of the dropped fault must be accumulated into the state differences of its next undetected fault. This process repeats until there are no test patterns or no undetected faults.

The problem with differential fault simulation is that the order of events caused by fault sites is not the same as the order of the timing of their occurrence. If the circuit behavior depends on the gate delay of the circuit, the timing

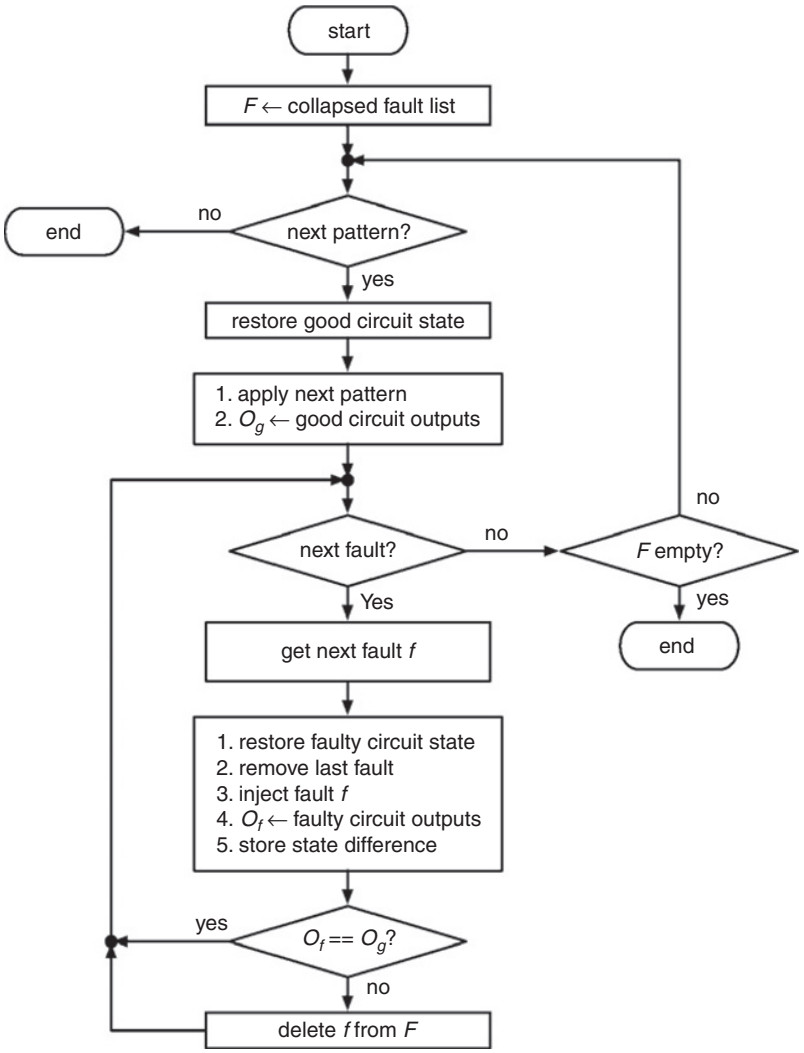



**FIGURE 14.19**

Differential fault simulation flowchart.

information of every event must be included. This solution, however, may potentially require high memory consumption.

### 14.3.5 Comparison of fault simulation techniques

In terms of simulation speed, it is apparent that serial fault simulation is the slowest among all the techniques. Differential fault simulation is shown to be up to twelve times faster than concurrent fault simulation and PPSFP [Cheng 1989], when the

sequential circuit under test does not contain memories, such as *static random-access memories* (SRAMs) and *dynamic random-access memories* (DRAMs).

Memory use is, in general, not a problem for serial fault simulation, because it deals with one fault at a time. Similarly, parallel fault simulation and PPSFP do not require much more memory than the fault-free simulation. Concurrent fault simulation has severe memory problems, because the size of the concurrent fault list is unpredictable. Furthermore, the I/O values of every bad gate in the concurrent fault simulation must be recorded. Differential fault simulation relieves the memory management problem of concurrent fault simulation, because only the difference in storage elements is stored.

When the unknown ($X$) and/or high-impedance ($Z$) values are present in the circuit, a multiple-valued fault simulation becomes necessary. Serial fault simulation has no problem in handling multiple-valued fault simulation, because it can be realized with a regular logic simulator. In contrast, to exploit bitwise word parallelism, it is more difficult for parallel fault simulation or PPSFP to handle $X$ or $Z$. In concurrent fault simulation, dealing with multiple-valued simulations is straightforward, because every bad gate is evaluated in the same way as in the fault-free simulation. Finally, differential fault simulation can simulate $X$ or $Z$ without a problem, because it is based on event-driven simulation.

From the aspect of delay and functional modeling capability, serial fault simulation does not encounter any difficulty. Parallel fault simulation and PPSFP cannot take delay or functional models into account, because they pack the information of multiple faults or test patterns into the same word and rely on bitwise logic operations. Being event-driven, both concurrent and differential fault simulation techniques are capable of handling functional models; however, only the former is able to process circuit delays.

When sequential circuits are of concern, serial and parallel fault simulation techniques do not have a problem. The PPSFP technique, however, is not suited for sequential circuit simulation, because a large memory space is required to store the states of the fault-free circuit. Concurrent and differential fault simulations are able to perform sequential fault simulation without difficulty.

On the basis of the previous discussions, PPSFP and parallel fault simulation techniques are currently the most popular fault simulation techniques for combinational (full-scan) circuits. On the other hand, concurrent fault simulation techniques have been widely adopted for sequential circuits embedded with memories, whereas differential fault simulation techniques are mostly suitable for sequential circuits without memories. Algorithm switching has also been used to improve performance. Parallel fault simulation can be used when the fault drop rate per test pattern is high, and then PPSFP is used when more patterns are required to drop each fault.

Even for fault simulation techniques that are efficient in time and memory, the problems of memory explosion and long simulation time still exist as *integrated circuit* (IC) complexity continues growing. To overcome the memory problem, the **multiple-pass fault simulation** approach is often adopted. The idea of

multiple-pass fault simulation is to partition the faults into smaller groups, each of which is simulated independently. If the faults are well partitioned, multiple-pass fault simulation prevents the memory explosion problem. To further reduce the fault simulation time, **distributed fault simulation** approaches may be used. Distributed fault simulation divides the whole fault simulation into smaller tasks, each of which is performed independently on a separate processor.

There are several alternatives to fault simulation. The fault-sampling technique was proposed to simulate only a sampled group of faults [Butler 1974]. Critical path tracing is another alternative to fault simulation [Abramovici 1984]. Instead of performing actual fault simulation, the ***statistical fault analysis*** (STAFAN) approach proposes to use probability theory to estimate the expected value of fault coverage [Jain 1985]. These alternatives to fault simulation have also been extensively discussed in [Abramovici 1994], [Bushnell 2000], and [Wang 2006].

## 14.4 TEST GENERATION

First, consider the single stuck-at fault model. Figure 14.20 shows a circuit with a single stuck-at fault in which signal *d* is tied to logic 1 (*d*/1). A logic 0 must be applied to node *d* from the primary inputs of the circuit to produce a difference between the fault-free (or good) circuit and the circuit with the stuck-at fault present. Next, to observe the effect of the fault, a logic 1 must be applied to signal *c*. So, if the fault *d*/1 is present, it can be detected at the output *e* with the derived vector. Test generation attempts to generate test vectors for every possible fault in the circuit. In this example, in addition to the *d*/1 fault, faults such as *a*/1, *b*/1, and *e*/1 are also targeted by the test generator. Because some of the faults in the circuit can be logically equivalent, no test can be obtained to distinguish between them. Thus, equivalence fault collapsing as described in Section 14.2 is often used to identify equivalent faults *a priori* to reduce the number of faults that must be targeted [Abramovici 1994; Bushnell 2000; Jha 2003]. Subsequently, the ATPG is only concerned with generating test vectors for each fault in the collapsed fault list.

### 14.4.1 Random test generation

***Random test generation*** (RTG) is one of the simplest methods for generating vectors. Vectors are randomly generated and fault-simulated (or fault-graded) on the ***circuit under test*** (CUT). Because no specific fault is targeted, the

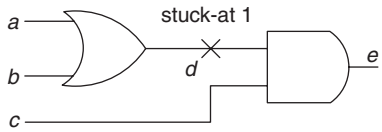



**FIGURE 14.20**

Example of a single stuck-at fault.

complexity of RTG is low. However, RTG often results in generating a large number of tests that achieves sub-par fault coverage because of the difficult-to-test faults.

In RTG, logic values are randomly generated at the primary inputs, with equal probability of assigning a logic 1 or logic 0 to each primary input. Thus, the random vectors are uniformly distributed in the test set. Note that the random test set is not truly random, because a pseudo-random number generator is generally used. In other words, the random test set can be repeated with the same pseudo-random number generator. Nevertheless, the vectors generated hold the necessary statistical properties of a random vector set.

The **level of confidence** one can have on a random test set $T$ can be measured as the probability that $T$ can detect all the stuck-at faults in the circuit. For $N$ random vectors, the **test quality** $t_N$ indicates the probability that all detectable stuck-at faults are detected by these $N$ random vectors. Thus, the test quality of a random test set highly depends on the circuit under test. Consider a circuit with an eight-input AND gate (or equivalently a cone of seven two-input AND gates) illustrated in Figure 14.21. Although achieving a logic 0 at the output of the AND gate is easy, getting a logic 1 is difficult. A logic 1 would require all the inputs to be at logic 1. If the RTG assigns each primary input with an equal probability of logic 0 or logic 1, the chance of getting eight logic 1's simultaneously would only be $0.5^8 = 0.0039$. In other words, the AND gate output stuck-at-0 fault would be difficult to test by the RTG. Such faults are called **random-pattern resistant faults**.

As discussed earlier, the quality of a random test set depends on the underlying circuit. More random-pattern resistant faults will more likely reduce the quality of the random test set. To tackle the problem of targeting random-pattern resistant faults, biasing is required so the input vectors are no longer viewed as uniformly distributed. Consider the same eight-input AND gate example again. If each input of the AND gate has a much higher probability of

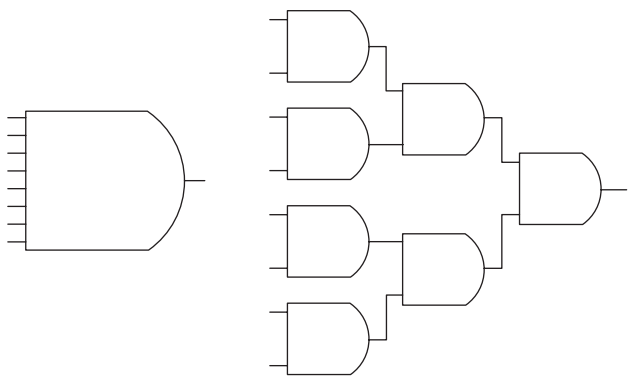


**FIGURE 14.21**

Two equivalent circuits.

receiving a logic 1, the probability of getting a logic 1 at the output of the AND gate significantly increases. For example, if each input has a 75% probability of receiving a logic 1, then getting a logic 1 at the output of the AND gate now becomes $0.75^8 = 0.1001$, rather than the previous 0.0039.

Determining the optimal bias values for each primary input that can achieve the highest coverage is not an easy task. Thus, rather than trying to obtain the best set of values, the objective is frequently to increase the probabilities for those difficult-to-control and difficult-to-observe nodes in the circuit. For instance, suppose a circuit has an eight-input AND gate; any fault that requires the AND gate output equal to logic 1 for detection will be considered difficult to test. It would then be beneficial to attempt to increase the probability of obtaining a logic 1 at the output of this AND gate.

Another issue regarding random test generation is the number of random vectors needed. Given a combinational circuit with $n$ primary inputs, there are clearly $2^n$ possible input vectors. One can express the probability of detecting fault $f$ by any random vector to be:

$$d_f = T_f / 2^n$$

where $T_f$ is the set of vectors that can detect fault $f$. Consequently, the probability that a random vector will not detect $f$ (*i.e.*, $f$ escapes a random vector) is: $e_f = 1 - d_f$.

Therefore, given $N$ random vectors, the probability that none of the $N$ vectors detect fault $f$ is:

$$e_f^N = (1 - d_f)^N$$

In other words, the probability that at least one of $N$ vectors will detect fault $f$ is:

$$1 - (1 - d_f)^N$$

If the detection probability, $d_f$, for the hardest fault is known, $N$ can be readily computed by solving the following inequality:

$$1 - (1 - d_f)^N \geq p$$

where $p$ is the probability that $N$ vectors should detect fault $f$.

If the detection probability is not known, it can be computed directly from the circuit. The detection probability of a fault is directly related to: (1) the controllability of the line that the fault is on and (2) the observability of the fault-effect to a primary output. The controllability and observability computations have been introduced previously in the chapter on design for testability. It is worth noting that the minimum detection probability of a detectable fault $f$ can be determined by the output cone in which $f$ resides. In fact, if $f$ is detectable, it must be excited and propagated to at least one primary output, as illustrated in Figure 14.22. It is clear that all the primary inputs necessary to excite $f$ and propagate the fault-effect must reside in the cone of the output

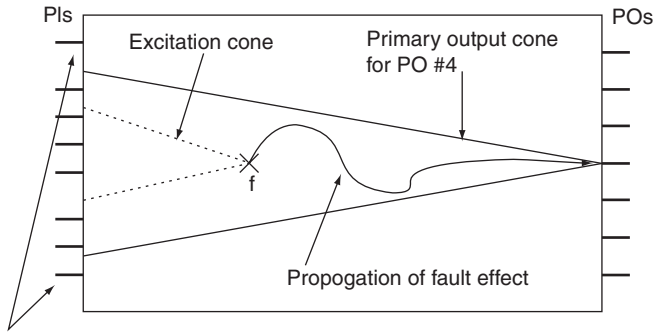



**FIGURE 14.22**

Detection of a fault.

to which $f$ is detected. Thus, the detection probability for $f$ is at least $(0.5)^m$, where $m$ is the number of primary inputs in the cone of the corresponding primary output. Taking this concept a step further, the detection probability of the most difficult fault can be obtained with the following lemma [David 1976; Shedletsky 1977].

*Lemma 1:* In a combinational circuit with multiple outputs, let $n_{max}$ be the number of primary inputs that can lead to a primary output. Then, the detection probability for the most difficult detectable fault, $d_{min}$, is:

$$d_{\min} \geq (0.5)^{nmax}$$

*Proof*

The proof follows from the preceding discussion.

#### 14.4.1.1 *Exhaustive testing*

If the combinational circuit has few primary inputs, **exhaustive testing** may be a viable option, where every possible input vector is enumerated. This may be superior to random test generation, because RTG can produce duplicated vectors and may miss certain ones.

In circuits in which the number of primary inputs is large, exhaustive testing becomes prohibitive. However, on the basis of the results of Lemma 1, it may be possible to partition the circuit and only exhaust the input vectors within each cone for each primary output. This is called **pseudo-exhaustive testing**. In doing so, the number of input vectors can be drastically reduced. When enumerating the input vectors for a given primary output cone, the values for the primary inputs that are outside the cone are simply assigned random values. Therefore, if a circuit has three primary outputs, each has a corresponding primary output cone. Note that these three primary output cones may overlap. Let $n1$, $n2$, and $n3$ be the number of primary inputs corresponding to these three cones. Then the number of pseudo-exhaustive vectors is simply at most $2^{n1} + 2^{n2} + 2^{n3}$.

### 14.4.2 Theoretical Background: Boolean difference

Consider the circuit shown in Figure 14.23. Let the target fault be the stuck-at-0 fault on primary input $y$. Recall the high-level concept of test generation illustrated in Figure 14.1, where the objective is to distinguish the fault-free circuit from the faulty circuit. In the example circuit shown in Figure 14.23, the faulty circuit is the circuit with $y$ stuck at 0. Note that the circuit output can be expressed as a Boolean formula:

$$f = xy + y'z$$

Let $f2$ be the faulty circuit with the fault y/0 present. In other words,

$$f2 = f(y = 0)$$

To distinguish the faulty circuit $f2$ from the fault-free counterpart $f$, any input vector that can make $f \oplus f2 = 1$ would suffice. Furthermore, because the aim is test generation, the target fault must be excited. In this example, the logic value on primary input $y$ must be logic 1 to excite the fault $y/0$. Putting these two conditions together, the following equation is obtained:

$$y \cdot f(y = 1) \oplus f(y = 0) = 1 \tag{14.1}$$

Note that $f(y = 1) \oplus f(y = 0)$ indicates the exclusive-or operation on the two functions $f(y = 1)$ and $f(y = 0)$; it evaluates to logic 1 if and only if the two functions evaluate to opposing values. In terms of ATPG, this is synonymous to propagating the fault effect at node $y$ to the primary output $f$. Therefore, any input vector on primary inputs $x$, $y$, and $z$ that can satisfy Equation (14.1) is a valid test vector for fault $y/0$:

$$y \cdot f(y = 1) \oplus f(y = 0) = y(x \oplus z) = y(xz' + x'z) = xyz' + x'yz$$

In this running example, the two vectors $xyz = \{110, 011\}$ are candidate test vectors for fault $y/0$. Formally, $f(y = 1) \oplus f(y = 0)$ is called the **Boolean difference** of $f$ with respect to $y$ and is often written as:

$$df/dy = f(y = 1) \oplus f(y = 0)$$

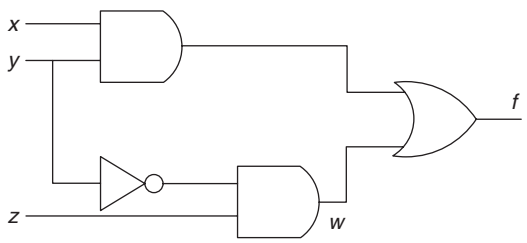



**FIGURE 14.23**

Example circuit to illustrate the concept of Boolean difference.

In general, if $f$ is a function of $x_1$, $x_2$, ... , $x_n$, then:

$$df/dx_i = f(x_1, x_2, \ldots, x_i = 1, \ldots, x_n) \oplus f(x_1, x_2, \ldots, x_i = 0, \ldots, x_n)$$

In terms of test generation, for any target fault on some fault $\alpha/v$, the set of all vectors that can *propagate* the fault-effect to the primary output $f$ is then those vectors that can satisfy:

$$df/d\alpha = 1$$

(Note that this is independent of the polarity of the fault, whether it is stuck-at-0 or stuck-at-1.) Next, the constraint that the fault must be excited, $\alpha$ set to value $v'$, must be added. Subsequently, the set of test vectors that can detect the fault becomes all those input values that can satisfy the following equation:

$$(\alpha = v') \cdot df/d\alpha = 1 \tag{14.2}$$

Consider the same circuit shown in Figure 14.23 again. Suppose the target fault is $w/0$. The same analysis can be performed for this new fault. The set of test vectors that can detect $w/0$ is simply:

$$\begin{aligned} & w \cdot df/dw = 1 \\ \Rightarrow & w \cdot f(w = 1) \oplus f(w = 0) = 1 \\ \Rightarrow & w \cdot (1 \oplus xy) = 1 \\ \Rightarrow & w \cdot (xy)' = 1 \\ \Rightarrow & w \cdot (x' + y') = 1 \\ \Rightarrow & wx' + wy' = 1 \end{aligned}$$

Now, $w$ can be expanded from the circuit shown in the figure to be $w = y' \cdot z$. Plugging this into the equation above gives us:

$$\begin{aligned} & w \cdot x' + w \cdot y' = 1 \\ \Rightarrow & \mathrm{y}' \cdot \mathrm{zx}' + \mathrm{y}' \cdot \mathrm{z} \cdot \mathrm{y} = 1 \\ \Rightarrow & \mathrm{x}' \cdot \mathrm{y}'\mathrm{z} + \mathrm{y}' \cdot \mathrm{z} = 1 \\ \Rightarrow & \mathrm{y}' \cdot \mathrm{z} = 1 \end{aligned}$$

Therefore, the set of vectors that can detect $w/0$ is {001, 101}.

#### 14.4.2.1 ***Untestable faults***

If the target fault is untestable, it would be impossible to satisfy Equation (14.2). Consider the circuit shown in Figure 14.24. Suppose the target fault is $z/0$. Then the set of vectors that can detect z/0 are those that can satisfy:

$$\begin{aligned} & z \cdot df/dz = 1 \\ \Rightarrow \quad & z \cdot f(z = 1) \oplus f(z = 0) = 1 \\ \Rightarrow \quad & z \cdot (xy \oplus xy) = 1 \\ \Rightarrow \quad & z \cdot 0 = 1 \\ \Rightarrow \quad & \text{UNSATISFIABLE} \end{aligned}$$

In other words, there exists no input vectors that can satisfy $z \cdot df/dz = 1$, indicating that the fault $z/0$ is untestable.

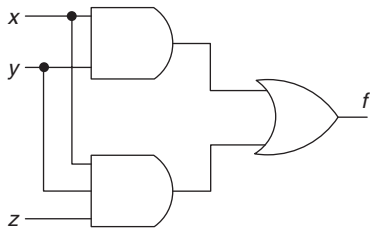



**FIGURE 14.24**

Example circuit for an untestable fault.

### 14.4.3 Designing a stuck-at ATPG for combinational circuits

In deterministic ATPG algorithms, there are two main tasks. The first is to excite the target fault, and the second is to propagate the fault-effect to a primary output. Because the logic values in both the fault-free and faulty circuits are needed, composite logic values are used. For each signal in the circuit, the values $v/v_f$ are needed, where $v$ denotes the value for the signal in the fault-free circuit, and $v_f$ represents the value in the corresponding faulty circuit. Whenever $v = v_f$, $v$ is sufficient to denote the signal value. To facilitate the manipulation of such composite values, a 5-valued algebra was proposed [Roth 1966], in which the five values are 0, 1, $X$, $D$, and $\bar{D}$; 0, 1, and $X$ are the conventional values found in logic design for true, false, and "don't care." $D$ represents the composite logic value 1/0 and $\bar{D}$ represents 0/1. Boolean operators such as AND, OR, NOT, and XOR can work on the 5-valued algebra as well. The simplest way to perform Boolean operations is to represent each composite value into the $v/v_f$ form and operate on the fault-free value first, followed by the faulty value. For example, 1 AND $D$ is 1/1 AND 1/0. AND-ing the fault-free values yields 1 AND 1 = 1, and AND-ing the faulty values yields 1 AND 0 = 0. So the result of the AND operation is 1/0 = $D$. As another example,

$$\begin{aligned} D \text{ OR } \bar{D} &= 1/0 \text{ OR } 0/1 \\ &= 1/1 \\ &= 1 \end{aligned}$$

Tables 14.6, 14.7, and 14.8 show the AND, OR, and NOT operations for the 5-valued algebra, respectively. Operations on other Boolean conjunctives can be constructed in a similar manner.

#### 14.4.3.1 *A naive ATPG algorithm*

A very simple and naive ATPG algorithm is shown in Algorithm 14.3, in which combinational circuits with fanout structures can be handled.

**Table 14.6** AND Operation

| **AND** | **0** | **1** | ***D*** | $\overline{D}$ | ***X*** |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | D | $\overline{D}$ | *X* |
| *D* | 0 | *D* | *D* | 0 | *X* |
| $\overline{D}$ | $\overline{D}$ | 1 | 1 | $\overline{D}$ | *X* |
| X | X | 1 | X | X | *X* |

**Table 14.7** OR Operation

| **OR** | **0** | **1** | ***D*** | $\overline{D}$ | ***X*** |
|---|---|---|---|---|---|
| 0 | 0 | 1 | *D* | $\overline{D}$ | *X* |
| 1 | 1 | 1 | 1 | 1 | 1 |
| *D* | *D* | 1 | *D* | 1 | *X* |
| $\overline{D}$ | $\overline{D}$ | 1 | 1 | $\overline{D}$ | *X* |
| X | *X* | 1 | X | *X* | *X* |

**Table 14.8** NOT Operation

| **NOT** | |
|---|---|
| 0 | 1 |
| 1 | 0 |
| *D* | $\overline{D}$ |
| $\overline{D}$ | *D* |
| *X* | *X* |

**Algorithm 14.3** Naive ATPG (*C*, *f* )

1. **while** a fault-effect of *f* has not propagated to a PO and all possible vector combinations have not been tried **do**
2. pick a vector, *v*, that has not been tried;
3. fault simulate *v* on the circuit *C* with fault *f*;
4. **end while**

Note that in an ATPG, the worst-case computational complexity is exponential, because all possible input patterns may have to be tried before a vector is found or that the fault is determined to be undetectable. One may go about line #2 of the algorithm in an intelligent fashion, so a vector is not simply selected indiscriminately. Whether or not intelligence is incorporated, some mechanism is needed to account for those attempted input vectors so no vector would be repeated. If it is possible to deduce some knowledge during the search for the input vector, the ATPG may be able to mark a set of solutions as tried and thus reduce the remaining search space. For instance, after attempting a number of input vectors, this naive ATPG realizes that any input vector with the first primary input set to logic 0 cannot possibly detect the target fault, and it can safely mark all vectors with the first primary input equal to 0 as a tried input vector. Subsequently, only those vectors with the first primary input set to 1 will be selected.

In certain cases, it may not be possible for the ATPG to deduce that all vectors with a given primary input set to some logic value would definitely not qualify to be solution vectors. However, it may be able to make an intelligent guess that input vectors with primary input *#i* set to some specific logic value are more likely to lead to a solution. In such a case, the ATPG would make a **decision** on primary input *#i*. Because the decision may actually be wrong, the ATPG may eventually have to alter its decision, trying the vectors that have the opposite Boolean value on primary input *#i*.

The process of making decisions and reversing decisions will result in a **decision tree**. Each node in the decision tree represents a decision variable. If only two choices are possible for each decision variable, then the decision tree is a binary tree. However, there may be cases in which multiple choices are possible in a general search tree.

Figure 14.25 shows an example decision tree. Although this figure only allows decisions to be made at the primary inputs, in general, this may not be

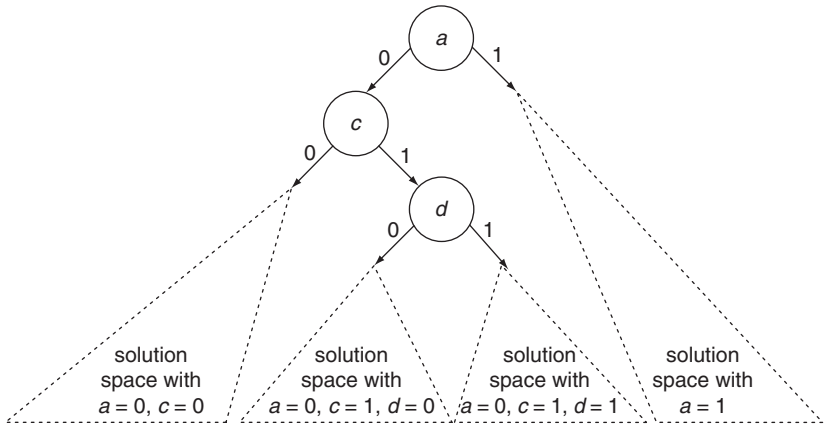



**FIGURE 14.25**

An example decision tree.

the case. This is used simply to allow the reader to have a clearer picture of the concept behind decision trees. At each decision, the search space is halved. For example, if the circuit has $n$ primary inputs, then there are a total of $2^n$ possible vectors in the solution space. After a decision is made, the solution spaces under the two branches of a decision node are disjoint. For instance, the space under the decision $a = 1$ does not contain any vectors with $a = 0$. Note that the decision tree for a solution vector may not require the ATPG to *exhaustively* enumerate every possible vector; rather, it *implicitly* enumerates the vectors. If a solution vector exists, there must be a path along the decision tree that leads to the solution. On the other hand, if the fault is undetectable, every path in the decision tree would lead to no solution. It is important to note that a fault may be detected without having made all decisions. For example, the circuit nodes that do not play a role in exciting or propagating the fault would not have to be included in the decision process. Likewise, it may not require all decision variables before the ATPG can determine that it is on the wrong path. For example, if a certain path already sets a value on the fault site such that the fault is not excited, then no value combination on the remaining decision variables can help to excite and propagate the fault. With Figure 14.25 as an example again, suppose the path $a = 0$, $c = 1$, $d = 1$ cannot excite the target fault $\alpha$. Then, the rest of the decision variables, $b$, $e$, $f$, ... , cannot undo the effect rendered by $a = 0, c = 1, d = 1$.

#### 14.4.3.1.1 Backtracking

Whenever a conflict is encountered (*i.e.*, a path segment in the decision tree leading to no solution), the search must not continue searching beneath that path but must go back to some earlier point and re-decide on a previous decision. If only two choices are possible for a decision variable, then some previous decision needs to be reversed if the other branch has not been explored before. This reversal of decision is called a **backtrack**. To keep track of where the search spaces have been explored and avoid repeating the search in the same

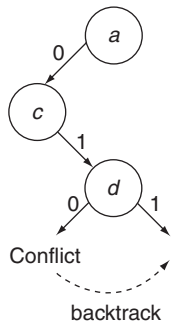



**FIGURE 14.26**

Backtrack on a decision.

spaces, the easiest mechanism is to reverse the most recent decision made. When reversing any decision, the signal values implied by the assignment of the previous decision variable must be undone.

Consider the decision tree illustrated in Figure 14.26 as an example. Suppose the current decisions made so far are $a = 0$, $c = 1$, $d = 0$, and this causes a conflict in detecting the target fault. Then, the search must reverse the most recently made decision, which is $d = 0$. When reversing $d = 0$ to $d = 1$, all values that resulted from $d = 0$ must be first undone. Then, the search continues with the path $a = 0$, $c = 1$, $d = 1$. If the reversal of a decision also caused a conflict (in this case, reversing $d = 0$ also caused a conflict), then it means $a = 0$, $c = 1$ actually cannot lead to any solution vector that can detect the target fault. The backtracking mechanism would then take the search to the previous decision and attempt to reverse that decision. In the running example, it would undo the decision on $d$, assigning $d$ to "don't care," followed by reversing of the decision $c = 1$ and searching the portion of the search space under $a = 0$, $c = 0$. Finally, if there is no previous decision that can be reversed, the ATPG concludes that the target fault is undetectable.

Technically, whenever a decision is reversed, say $d = 0$ is reversed to $d = 1$ as shown in Figure 14.26, $d = 1$ is no longer a decision; rather, it becomes an implied value by a subset of the previous decisions made. The exact subset of decisions that implied $d = 1$ can be computed by a **conflict analysis** [Marques-Silva 1999b]. However, the details of conflict analysis are beyond the scope of this chapter and are thus omitted. The reader can refer to [Marques-Silva 1999b] for details of this mechanism. In addition, intelligent conflict analysis can also allow for **nonchronological backtracking**.

#### 14.4.3.2 *A basic ATPG algorithm*

Given a target fault $g/v$ in a fanout-free combinational circuit $C$, a simple procedure to generate a vector for the fault is shown in Algorithm 14.4, where JustifyFanoutFree() and PropagateFanoutFree() are both recursive functions.

**Algorithm 14.4** Basic Fanout Free ATPG ($C$, $g/v$)

1. initialize circuit by setting all values to $X$;
2. JustifyFanoutFree($C$, $g$, $v'$); /* excite the fault by justifying line $g$ to $v'$ */
3. PropagateFanoutFree($C$, $g$); /* propagate fault-effect from $g$ to a PO */

The JustifyFanoutFree($g$, $v$) function recursively justifies the predecessor signals of $g$ until all signals that need to be justified are, indeed, justified from the primary inputs. The simple outline of the JustifyFanoutFree routine is listed in Algorithm 14.5. In line #10 of the algorithm, controllability measures can be used to select the best input to justify. Selecting a good gate input may help to reach a primary input sooner.

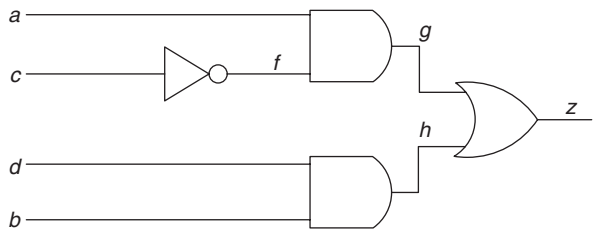



**FIGURE 14.27**

Example fanout-free circuit.

Consider the circuit $C$ shown in Figure 14.27. Suppose the objective is to justify $g = 1$. According to the preceding algorithm, the following sequence of recursive calls to JustifyFanoutFree would have been made:

call #1: JustifyFanoutFree($C$, $g$, 1)
call #2: JustifyFanoutFree($C$, $a$, 1)
call #3: JustifyFanoutFree($C$, $f$, 1)
call #5: JustifyFanoutFree($C$, $c$, 0)

**Algorithm 14.5** JustifyFanoutFree($C$, $g$, $v$)

```
1. g = v;
2. if gate type of g == primary input then
3.    return;
4. else if gate type of g == AND gate then
5.  if v == 1 then
6.       for all inputs h of g do
7.          JustifyFanoutFree(C, h, 1);
8.       end for
9.  else {v == 0}
10.     h = pick one input of g whose value == X;
11.     JustifyFanoutFree(C, h, 0);
12.   end if
13. else if gate type of g == OR gate then
14. ...
15. end if
```

After these calls to JustifyFanoutFree(), $abcd = 1X0X$ is an input vector that can justify $g = 1$.

Consider another circuit $C$ shown in Figure 14.28. Note that the circuit is not fanout-free, but the preceding algorithm will still work for the objective of

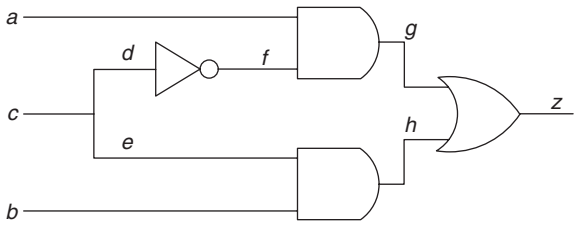



**FIGURE 14.28**

Example circuit with a fanout structure.

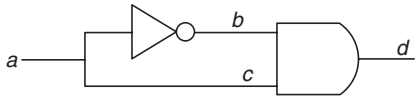



**FIGURE 14.29**

Circuit with a constant circuit node.

trying to justify the signal $g = 1$. According to the algorithm, the following sequence of calls to the JustifyFanoutFree function would have been made:

call #1: JustifyFanoutFree($C$, $g$, 1)
call #2: JustifyFanoutFree($C$, $a$, 1)
call #3: JustifyFanoutFree($C$, $f$, 1)
call #4: JustifyFanoutFree($C$, $d$, 0)
call #5: JustifyFanoutFree($C$, $c$, 0)

After these five calls to JustifyFanoutFree(), $abc = 1X0$ is an input vector that can justify $g = 1$. Note that in a fanout-free circuit, the JustifyFanoutFree() routine will *always* be able to set $g$ to the desired value $v$, and no conflict will ever be encountered. However, this is not always true for circuits with fanout structures, such as the circuit shown in Figure 14.29. This is because in circuits with fanout branches, two or more signals that can be traced back to the same fanout stem are **correlated**, and setting arbitrary values on these correlated signals may not always be possible. For example, in the simple circuit shown in Figure 14.29, justifying $d = 1$ is impossible, because it requires both $b = 1$ and $c = 1$, thereby causing a conflict on $a$.

Consider again the circuit shown in Figure 14.28. Suppose the objective is to set z = 0. On the basis of the JustifyFanoutFree() algorithm, it would first justify both $g = 0$ and $h = 0$. Now, for justifying $g = 0$, suppose it picks the signal $f$ for justifying the objective $g = 0$; it would eventually assign $c = 1$ through the recursive JustifyFanoutFree() function. Next, for justifying $h = 0$, it no longer can choose $e = 0$ as a viable option, because choosing e = 0 will eventually cause a **conflict** on signal $c$. In other words, a different **decision** has to be made for justifying $h = 0$. In this case, $b = 0$ should be chosen. Although this example is very simple, it illustrates the possibility of making poor decisions, causing potential **backtracks** in the search. In the rest of this chapter, more discussion on avoiding conflicts will be covered.

In the preceding running example, suppose the target fault is $g/0$, and JustifyFanoutFree($C$, $g$, 1) would have successfully excited the fault. With the fault $g/0$ excited, the next step is to propagate the fault-effect to a primary output. Similar to the JustifyFanoutFree() function, PropagateFanoutFree() is a recursive function as well, where the fault-effect is propagated one gate at a time until it reaches a primary output. Algorithm 14.6 illustrates the pseudo-code for one possible implementation of the propagate function.

Again, although the PropagateFanoutFree() routine is meant for fanout-free circuits, it is sufficient for the running example. With the PropagateFanoutFree() function on the fault-effect $D$ at signal $g$, listed in Algorithm 14.5, the following calls to the JustifyFanoutFree and PropagateFanoutFree functions would have been made:

call #1: PropagateFanoutFree($C$, $g$)
call #2: JustifyFanoutFree($C$, $b$, 0)
call #3: JustifyFanoutFree($C$, $b$, 0)
call #4: PropagateFanoutFree($C$, $z$)

**Algorithm 14.6** PropagateFanoutFree($C$, $g$)

```
1. if g has exactly one fanout then
2.    h = fanout gate of g;
3.    if none of the inputs of h has the value of X then
4.       backtrack;
5.    end if
6. else {g has more than one fanout}
7.    h = pick one fanout gate of g that is unjustified;
8. end if
9. if gate type of h == AND gate then
10.  for all inputs, j, of h, such that j ≠ g do
11.    if the value on j == X then
12.       JustifyFanoutFree(C, j, 1);
13.    end if
14.  end for
15. else if gate type of h == OR gate then
16.  for all inputs, j, of h, such that j ≠ g do
17.    if the value on j == X then
18.       JustifyFanoutFree(C, j, 0);
19.    end if
20.  end for
21. else if gate type of h == ... gate then
22. ...
23. end if
24. PropagateFanoutFree(C, h);
```

Because the fault-effect has successfully propagated to the primary output $z$, the fault $g/0$ is detected, with the vector $abc = 100$. The reader may notice that once $g/0$ has been excited, it is also propagated to $z$ as well, because $c = 0$ also has made $b = 0$. In other words, the JustifyFanoutFree($C$, $b$, 0) step is unnecessary. However, this is only possible if logic simulation or implication capability is embedded in the BasicFanoutFreeATPG() algorithm. For this discussion, it is not assumed that logic simulation is included.

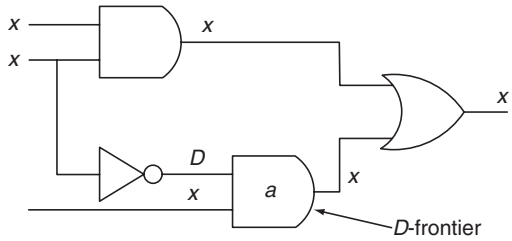
With the same circuit shown in Figure 14.28, consider the fault $g/1$. The BasicFanoutFreeATPG() algorithm will again be used to generate a test vector for this fault. In this case, the ATPG first attempts to justify $g = 0$, followed by propagating the fault-effect to $z$. During the justification of $g = 0$, the ATPG can pick either $a$ or $f$ as the next signal to justify. At this point, the ATPG must make a **decision**. Testability measures discussed in an earlier chapter can be used as a guide to make more intelligent decisions. In this example, choosing $a$ is considered to be better than $f$, because choosing $a$ requires no additional decisions to be made. Note that testability measures only serve as a guide to decision selection; they do not guarantee that the guidance will always lead to better decision selection.

It is important to note that in circuits with fanout structures, because the simple JustifyFanoutFree() and PropagateFanoutFree() functions described previously are meant for fanout-free circuits, will not always be applicable as illustrated in some of the earlier examples because of potential conflicts. To generate test vectors for general combinational circuits, there must be mechanisms that will allow the ATPG to avoid conflicts, as well as get out of a conflict when a conflict is encountered. To do so, the corresponding decision tree must be constructed during the search for a solution vector, and backtracks must be enforced for any conflict encountered. The following sections describe a few ATPG algorithms.

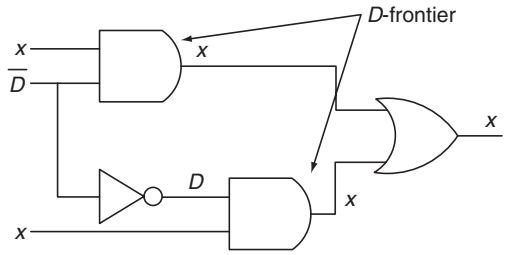
#### 14.4.3.3 *D algorithm*

The $D$ algorithm was proposed to tackle the generation of vectors in general combinational circuits [Roth 1966, 1967]. As indicated by the name of the algorithm, the $D$ algorithm tries to propagate a $D$ or $\overline{D}$ of the target fault to a primary output. Initially, every signal in the circuit has the unknown value, X. At the end of the $D$ algorithm, some signals will be assigned 0, 1, $D$, or $\overline{D}$, while the rest of the signals may remain as unknown. Note that because each detectable fault can be excited, a fault-effect can always be created. In the following discussion, propagation of the fault-effect will take precedence over the justification of the signals. This allows for enhanced efficiency of the algorithm and for simpler discussion.

Before proceeding to discussing the details of the $D$ algorithm, two important terms should be defined: the ***D*-frontier** and the ***J*-frontier**. The $D$-frontier consists of all the gates in the circuit whose output value is unspecified and a fault-effect ($D$ or $\overline{D}$) is at one or more of its inputs. For this to occur, one or more inputs of the gate must currently have an unknown value, X. For example, at the start of the $D$ algorithm, for a target fault $f$ there is exactly one $D$ (or $\overline{\mathrm{D}}$) placed in the circuit




(a) *D*-frontier contains one gate




(b) *D*-frontier contains two gates

**FIGURE 14.30**

Illustrations of *D*-frontier.

corresponding to the stuck-at fault. All other signals currently have a "don't care" value. Thus, the *D*-frontier consists of the successor gate(s) from the line with the fault *f*. Two scenarios of a *D*-frontier are illustrated in Figure 14.30. Clearly, at any time if the *D*-frontier is empty, the fault no longer can be detected. For example, consider Figure 14.30a. If the bottom input of gate *a* is assigned a value of 0, the output of gate *a* will become 0, and the D-frontier now becomes empty. At this time, the search must backtrack and try a different search path.

The *J*-frontier consists of all the gates in the circuit whose output values are known (can be any value in the 5-valued logic except X) but is not justified by its inputs. Figure 14.31 illustrates an example of a *J*-frontier. Thus, to detect the target fault, all gates in the *J*-frontier must be justified; otherwise, some gates in the *J*-frontier must have caused a conflict, where these gates cannot be justified to the desired values.

Having discussed the two fundamental concepts of the *D*-frontier and the *J*-frontier, the explanation for the *D* algorithm can begin. The *D* algorithm begins by trying to propagate the initial *D* (or $\overline{D}$) at the fault site to a primary output. For example, in Figure 14.32, the propagation routine will set all the side inputs of the path necessary (gates $a \rightarrow b \rightarrow c$) to propagate the fault-effect to the respective noncontrolling values. These side input gates, namely *x*, *y*, and *z*, thus form the *J*-frontier, because they are not currently justified. Because the *D* is propagated to the primary output, the *D*-frontier eventually becomes the output gate.

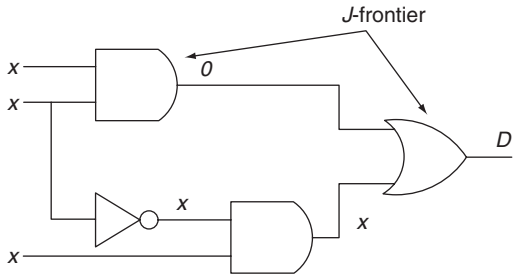



**FIGURE 14.31**

Illustration of *J*-frontier.

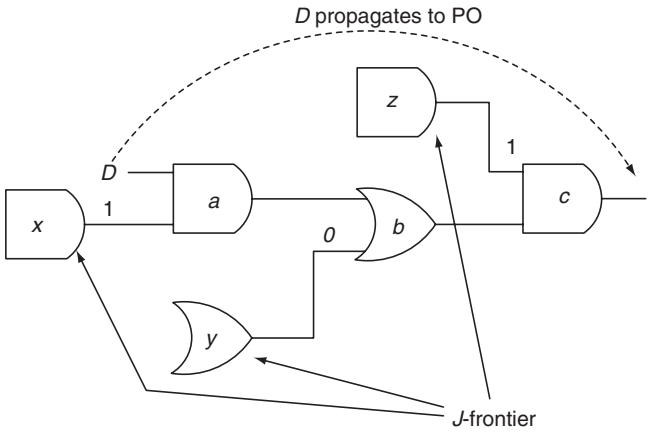



**FIGURE 14.32**

Propagation of *D*- and *J*-frontier.

Whenever there are paths to choose from in advancing the *D*-frontier, observability values can be used to select the corresponding gates. However, this does not guarantee that the more observable path will definitely lead to a solution. When a $D$ or a $\overline{D}$ has reached a primary output, all the gates in the *J*-frontier must now be justified. This is done by advancing the *J*-frontier backward by placing predecessor gates in the *J*-frontier such that they justify the previous unjustified gates. Similar to propagation of the fault-effect, whenever a conflict occurs, a backtrack must be invoked. In addition, at each step, the *D*-frontier must be checked so the $D$ (or $\overline{D}$) that has reached a primary output is still there. Otherwise, the search returns to the propagation phase and attempts to propagate the fault-effect to a primary output again. The overall procedure for the $D$ algorithm is shown in Algorithms 14.7 and 14.8.

Note that the previous procedure has not incorporated any intelligence in the decision-making process. In other words, sometimes it may be possible to determine that some value assignments are not justifiable, given the current

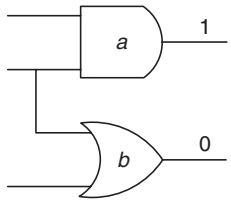



**FIGURE 14.33**

Conflict in the justification process.

circuit state. For instance, consider the circuit fragment shown in Figure 14.33. Justifying gate $a = 1$ and gate $b = 0$ is not possible, because $a = 1$ requires both of its inputs set to logic 1, whereas $b = 0$ requires both of its inputs set to logic 0. Noting such conflicting scenarios early can help to avoid future backtracks. Such knowledge can be incorporated into line #1 of the *D*-Alg-Recursion() shown in Algorithm 14.8. In particular, implications can be used to identify such potential conflicts, and they are used extensively to enhance the performance of the *D* algorithm (as well as other ATPG algorithms).

**Algorithm 14.7** *D*-Algorithm(*C*, *f* )

1. initialize all gates to don't-cares;
2. set a fault-effect ($D$ or $\overline{D}$) on line with fault f and insert it to the *D*-frontier;
3. *J*-frontier = $\phi$;
4. result = *D*-Alg-Recursion(*C*);
5. **if** result == success **then**
6. print out values at the primary inputs;
7. **else**
8. print fault *f* is untestable;
9. **end if**

Consider the multiplexer circuit shown in Figure 14.28. If the target fault is *f* stuck-at-0, then, after initializing all gate values to *X*, the D algorithm places a D on line *f*. The algorithm then tries to propagate the fault-effect to *z*. First, it will place $a = 1$ in the *J*-frontier, followed by $b = 0$ in the *J*-frontier. At this time, the fault-effect has reached the primary output. Now, the ATPG tries to justify all unjustified values in the *J*-frontier. Because *a* is a primary input, it is already justified. The other signals in the *J*-frontier are $f = D$ and $b = 0$. For $f = D$, $d = 0$, thereby making $c = 0$. For $b = 0$, either $e = 0$ or $b = 0$ is sufficient. Whichever one it picks, the search process will terminate, as a solution has been found.

Consider the same multiplexer circuit (see Figure 14.28) again. Suppose the target fault now is *f* stuck-at-1. Following the similar discussion as the previous target fault *f*/0, the algorithm initializes the circuit and places a *D* on *f*. Next, to propagate the fault-effect to a primary output, it likewise inserts a = 1 and h = 0 into the *J*-frontier. Now, the ATPG needs to justify all the gates in the J-frontier,

**Algorithm 14.8** D-Alg-Recursion($C$)

```
1. if there is a conflict in any assignment or D-frontier is φ then
2.    return failure
3. end if
4. /* first propagate the fault-effect to a PO */
5. if no fault-effect has reached a PO then
6.   while not all gates in D-frontier has been tried do
7.    g = a gate in D-frontier that has not been tried;
8.    set all unassigned inputs of g to non-controlling value and add them
      to the J-frontier;
9.    result = D-Alg-Recursion(C);
10.   if result == success then
11.     return (success);
12.   end if
13.  end while
14. return (failure);
15. end if {fault-effect has reached at least one PO}
16. if J-frontier is φ then
17.    return (success);
18. end if
19. g = a gate in J-frontier;
20. while g has not been justified do
21.    j = an unassigned input of g;
22.    set j = 1 and insert j = 1 to J-frontier;
23.    result = D-Alg-Recursion(C);
24.    if result == success then
25.       return (success);
26.    else try the other assignment
27.       set j = 0;
28.    end if
29. end while
30. return(failure);
```

which includes $a = 1$, $f = D$, and $b = 0$. Because $a$ is a primary output, it is already justified. For $f = D$, $d = 1$. For $b = 0$, suppose it selects $e = 0$. At this time, the $J$-frontier consists of two gate values: $d = 1$ and $e = 0$. No value assignment on $c$ can satisfy both $d = 1$ and $e = 0$; therefore, a conflict has occurred, and backtrack on the previous decision is needed. The only decision that has been made is $e = 0$ for $b = 0$, because there were two choices possible for

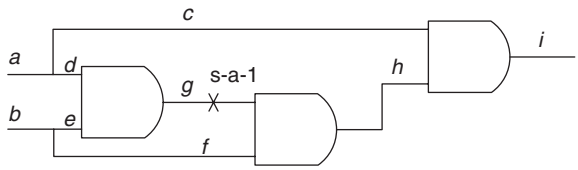



**FIGURE 14.34**

Example circuit.

justifying $h = 0$. At this time, the value on e is reversed, and $b = 0$ is added to the *J*-frontier. The process continues and all gate values in the *J*-frontier can be successfully justified, ending the process with the vector $abc = 101$.

Note that, in the preceding example, if some learning procedure (such as implications) is present, the decision for $h = 0$ would not result in $e = 0$, because the ATPG would have detected that $e = 0$ would conflict with $d = 1$. This knowledge could potentially improve the performance of the ATPG, which will be discussed later in this chapter.

Consider another example circuit shown in Figure 14.34. Suppose the target fault is $g/1$. After circuit initialization, the *D* algorithm places a $\overline{D}$ on $g$. Now, the *J*-frontier consists of $g = \overline{D}$ and the *D*-frontier consists of $h$. To advance the *D*-frontier, $f$ is set to logic 1; $f = 1$ is added to the *J*-frontier, and the *D*-frontier is now $i$. Next, to propagate the fault-effect to the output, $c = 1$ is added to the *J*-frontier. At this time, the fault-effect has been propagated to the output, and the task is to justify the signal values in the *J*-frontier: $\{g = \overline{D}, f = 1, c = 1\}$. To justify $g = \overline{D}$, two choices are possible: $a = 0$ or $b = 0$. If $a = 0$ is selected, it is necessary to justify $f = 1$, $b = 1$. Finally, $c = 1$ remains in the *J*-frontier which is still unjustified. At this time, a contradiction has occurred ($a = 0$ and $c = 1$), and the search reverses its last decision, changing $a = 0$ to $a = 1$. The search discovers that this reversal also causes a conflict. Thus, a backtrack occurs where line $b$ is chosen instead of $a$ for the previous decision, so $a$ is reset to "don't care." By assigning $b = 0$, a conflict is observed. Reversing $b$ also cannot justify all the *J*-frontier. At this time, backtracking on $b$ leads to no prior decisions. Thus, target fault $g/1$ is declared to be untestable.

### 14.4.4 PODEM

In the *D* algorithm, the decision space encompasses the entire circuit. In other words, every internal gate could be a decision point. However, noting that the end result of any ATPG algorithm is to derive a solution vector at the primary inputs and that the number of primary inputs generally is much fewer than the total number of gates, it may be possible to arrive at a very different ATPG algorithm that makes decisions only at primary inputs rather than at internal nodes of the circuit.

The ***path-oriented decision-making*** (PODEM) algorithm [Goel 1981] is based on this notion and makes decisions only at the primary inputs. Similar

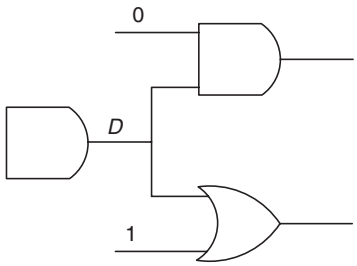



**FIGURE 14.35**

No *X* path.

to the *D* algorithm, a *D*-frontier is kept. However, because decisions are made at the primary inputs, the *J*-frontier is unnecessary. At each step of the ATPG search process, it checks whether the target fault is excited. If the fault is excited, it then checks whether there is an *X*-path from at least one fault- effect in the *D*-frontier to a primary output, where an *X*-path is a path of unspecified values from the fault-effect to a primary output. If no *X*-path exists, it means that all the fault-effects in the *D*-frontier are blocked, as illustrated in Figure 14.35, where both possible propagation paths of the *D* have been blocked. Otherwise, PODEM will pick the best *X*-path to propagate the fault-effect. Note that if the target fault has not been excited, the first steps of PODEM will be to excite the fault.

The basic flow of PODEM is illustrated in Algorithms 14.9 and 14.10. Although it is still a deterministic search algorithm, the decisions are limited to the primary inputs. All internal signals obtain their logic values by means of logic simulation (or implications) from the decision points. As a result, no conflict will ever occur at the internal signals of the circuit. The only possible conflicts in PODEM are either (1) the target fault is not excited or (2) the *D*-frontier becomes empty. In either of these cases, the search must backtrack.

**Algorithm 14.9** PODEM(*C*, *f*)

```
1. initialize all gates to don't-cares;
2. D-frontier = φ;
3. result = PODEM-Recursion(C);
4. if result == success then
5.    print out values at the primary inputs;
6. else
7.    print fault f is untestable;
8. end if
```

**Algorithm 14.10** PODEM-Recursion($C$)

```
1. if fault-effect is observed at a PO then
2.     return (success);
3. end if
4. (g, v) = getObjective(C);
5. (pi, u) = backtrace (g, v);
6. logicSimulate_and_imply (pi, u);
7. result = PODEM-Recursion(C);
8. if result == success then
9.     return(success);
10. end if
11. /* backtrack */
12. logicSimulate_and_imply (pi, ū);
13. result = PODEM-Recursion(C);
14. if result == success then
15.     return(success);
16. end if
17. /* bad decision made at an earlier step, reset pi */
18. logicSimulate_and_imply (pi, X);
19. return(failure);
```

According to the algorithm in PODEM, the search starts by picking an objective, and it backtraces from the objective to a primary input by means of the best path. Controllability measures can be used here to determine which path is regarded as the best. Gradually more primary inputs will be assigned logic values. At any time the target fault becomes unexcited or the *D*-frontier becomes empty, a bad decision must have been made, and reversal of some previous decisions is needed. The backtracking mechanism proceeds by reversing the most recent decision. If reversing the most recent decision also causes a conflict, the recursive algorithm will continue to backtrack to earlier decisions, until no more reversals are possible, at which time the fault is determined to be undetectable.

Three important functions in PODEM-Recursion() are getObjective(), backtrace(), and logicSimulate_and_imply(). The getObjective() function returns the next objective the ATPG should try to justify. Before the target fault has been excited, the objective is simply to set the line on which the target fault resides to the value opposite to the stuck value. Once the fault is excited, the getObjective() function selects the best fault-effect from the *D*-frontier to propagate. The pseudo-code for getObjective() is shown in Algorithm 14.11.

**Algorithm 14.11** getObjective(*C*)

1. **if** fault is not excited **then**
2. return ($g$, $\overline{v}$);
3. **end if**
4. $d$ = a gate in *D*-frontier;
5. $g$ = an input of $d$ whose value is $X$;
6. $v$ = noncontrolling value of $d$;
7. return ($g$, $v$);

The backtrace() function returns a primary input assignment from which there is a path of unjustified gates to the current objective. Thus, backtrace() will never traverse through a path consisting of one or more justified gates. From the objective's point of view, the getObjective() function returns an objective, say $g = v$, which means the current value of $g$ is unspecified and should be set to value $v$. If $g$ was already specified to $v$, $g = v$ would have never been selected as an objective, because it is already justified. Now, if $g = x$ currently, and the objective is to set $g = v$, there must exist a path of unjustified gates from at least one primary input to $g$. This backtrace() function can simply be implemented as a loop from the objective to some primary inputs through a path of unspecified values. Algorithm 14.12 shows the pseudo-code for the backtrace() routine.

Finally, the logicSimulate_and_imply() function can simply be a regular logic simulation routine. The added imply is used to derive additional implications, if any, that can enhance the getObjective() routine later on.

Consider the multiplexer circuit shown in Figure 14.28 again. Consider the target fault $f$ stuck-at-0. First, PODEM initializes all gate values to $X$. Then, the first objective would be to set $f = 1$. The backtrace routine selects $c = 0$ as the decision. After logic simulation, the fault is excited, together with $e = h = 0$. The *D*-frontier at this time is $g$. The next objective is to advance the *D*-frontier, thus getObjective() returns $a = 1$. Because $a$ is already a primary input, backtrace() will simply return $a = 1$. After simulating $a = 1$, the fault-effect is successfully propagated to the primary output $z$, and PODEM is finished with this target fault with the computed vector $abc = 1X0$. Table 14.9 shows the series of objectives and backtraces for this example.

**Table 14.9** PODEM Objectives and Decisions for *f* Stuck-At-0

| getObjective() | backtrace() | logicSim() | D-frontier |
|---|---|---|---|
| $f = 1$ | $c = 0$ | $d = 0$, $f = \mathrm{D}$, $e = 0$, $h = 0$ | $g$ |
| $a = 1$ | $a = 1$ | $g = D$, $z = D$ | $f$/0 detected |

**Algorithm 14.12** backtrace(*C*)

```
1. i = g;
2. num_inversion = 0;
3. while i ≠ primary input do
4.       i = an input of i whose value is X;
5.       if i is an inverted gate type then
6.          num_inversion ++;
7.       end if
8. end while
9. if num_inversion == odd then
10.      v = v̄;
11. end if
12. return (i, v);
```

Consider the circuit shown in Figure 14.29. Suppose the target fault is $b$ stuck-at-0. After circuit initialization, the first objective is $b = 1$ to excite the fault. The backtrace() returns $a = 0$. After logic simulation, although the target fault is excited, there is no *D*-frontier, because $c = d = 0$. At this time, PODEM reverses its last decision $a = 0$ to $a = 1$. After logic simulating $a = 1$, the target fault is not excited and the *D*-frontier is still empty. PODEM backtracks but there is no prior decision point. Thus, it concludes that fault $b/0$ is undetectable. Table 14.10 shows the steps made for this example, and Figure 14.36 shows the corresponding decision tree.

Consider again the circuit shown in Figure 14.34 with the target fault $g/1$. After circuit initialization, the first objective is to excite the fault; in other words, the objective is $g = 0$. The backtrace() function backtraces from the objective backward to a primary input via a path of "don't cares." Suppose the

**Table 14.10** PODEM Objectives and Decisions for $b$ Stuck-At-0

| getObjective() | backtrace() | logicSim() | D-frontier |
|---|---|---|---|
| b = 1 | a = 0 | b = 1, c = 0, d = 0 | {} |
| a = 1 (reversal) | – | b = 0, c = 1, d = 0 | {} |

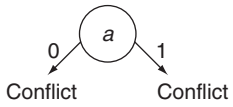



**FIGURE 14.36**

Decision tree for fault $b/0$.

backtrace reaches $a = 0$. After logic simulation, $g = 0$, $c = d = 0$, and $i = 0$. The *D*-frontier is $h$. However, note that there is no path of "don't cares" from any fault-effect in the *D*-frontier to a primary output! If the PODEM algorithm is modified to check that any objective has at least a path of "don't cares" to one or more primary outputs, some needless searches can be avoided. For instance, in this example, if the next objective was $f = 1$, even after the decision of $b = 1$ is made, the target fault still would not have been detected, because there was no path to propagate the fault-effect to a primary output even before the decision $b = 1$ was made. In other words, the search could immediately backtrack on the first decision $a = 0$. In this case, $a = 1$, and the objective is still $g = 0$. Backtrace() will now return $b = 0$. After logic simulation, $g = 0$, $c = 1, f = 0, h = 0, i = 0$. Again, there is no propagation path possible. As there is no earlier decision to backtrack to, the ATPG concludes that fault $g/1$ is untestable. Table 14.11 shows the steps for this example.

### 14.4.5 FAN

Although PODEM reduces the number of decision points from the number of gates in the circuit to the number of primary inputs, it still can make an excessive number of decisions. Furthermore, because PODEM targets one objective at a time, the decision process may sometimes be too localized and miss the global picture. The ***fanout-oriented test generation*** (FAN) algorithm [Fujiwara 1983] extends the PODEM-based algorithm to remedy these shortcomings.

To reduce the number of decision points, FAN first identifies the **headlines** in the circuit, which are the output signals of fanout-free regions. Because of the fanout-free nature of each cone, all signals outside the cone that do not conflict with the headline assignment would never require a conflicting value assignment on the primary inputs of the corresponding fanin cone. In other words, any value assignment on the headline can always be justified by its fanin cone. This allows the backtrace() function to backtrace to either headlines or primary inputs. Because each headline has a corresponding fanin cone with several primary inputs, this allows the number of decision points to be reduced.

Consider the circuit shown in Figure 14.37. If the current objective is to set $z = 1$, the corresponding decision tree based on the PODEM algorithm will involve many decisions at the primary inputs, such as $a = 1$, $c = 1$, $d = 1$, $e = 1$, $f = 1$. On the other hand, the decision based on the FAN algorithm is

**Table 14.11** PODEM Objectives and Decisions for *g* Stuck-At-1

| getObjective() | backtrace() | logicSim() | D-frontier |
|---|---|---|---|
| $g = 1$ | $a = 0$ | $g = D$, $c = 0$, d = 0, $i = 0$ | {*h*} (but no X-path to PO) |
| $a = 1$ (reversal) | – | $c = 1$, $d = 1$ | {} |

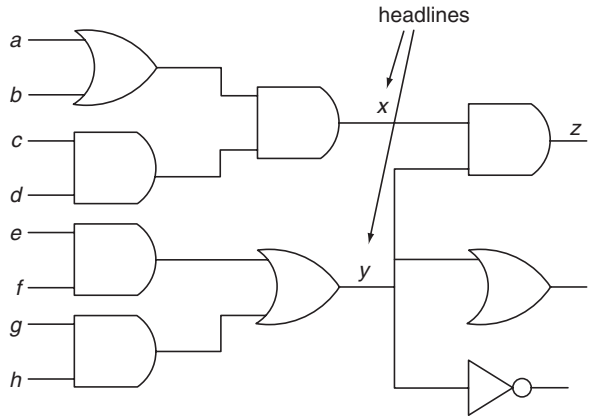



**FIGURE 14.37**

Circuit with identified headlines.

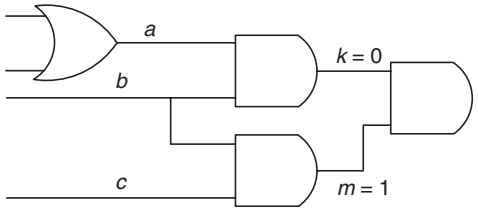



**FIGURE 14.38**

Multiple backtrace to avoid potential conflicts.

significantly smaller, involving only two decisions: $x = 1$ and $y = 1$. If $z = 1$ was not the first objective, there would have been other decisions made earlier. In other words, if there were a poor decision made in an earlier step, PODEM would need to reverse and backtrack many more decisions compared with FAN.

The next improvement that FAN makes over PODEM is the simultaneous satisfaction of multiple objectives, as opposed to only one target objective at each step. Consider the circuit fragment shown in Figure 14.38. Without taking into account multiple objectives, the backtrace() routine may choose the easier path in trying to justify $k = 0$. The easier path may be through the fanout stem $b$. However, this would cause a conflict later on with the other objective $m = 1$. In FAN, multiple objectives are taken into account, and the backtrace routine scores the nodes visited from each objective in the current set of objectives. The nodes along the path with the best scores are chosen. In this example, $a = 0$ will be chosen rather than $b = 0$, even if $a = 0$ is less controllable.

A powerful implication engine can have a significant impact on the performance of ATPG algorithms. Thus, much effort has been invested over the years in the efficient computation of implications. The quality of implications was

improved with the computation of indirect implications in **SOCRATES** [Schulz 1988]. **Static learning** was extended to **dynamic learning** in [Schulz 1989 and Kunz 1993], where some nodes in the circuit already had value assignments during the learning process. A 16-valued logic was introduced in [Rajski 1990 and Cox 1994]. Reduction lists were used to dynamically determine the gate values. In [Chakradhar 1993], the authors proposed a transitive closure procedure based on the implication graph. **Recursive learning** was later proposed in [Kunz 1994] in which a complete set of pairwise implications could be computed. To keep the computational costs low, a small recursion depth can be enforced in the recursive learning procedure. Finally, implications to capture time frame information in sequential circuits in a graphical representation were proposed in [Zhao 2001] to compactly store the implications in sequential circuits.

The implications can be used to quickly identify untestable faults [Iyer 1996a,b; Zhao 2001; Hsiao, 2002; Syal 2003]. This will allow the ATPG not to specifically target these faults that can often consume much of the ATPG computational resources. For more information on implication and untestable fault identification, refer to [Bushnell 2000, Jha 2003, and Wang 2006].

## 14.5 ADVANCED TEST GENERATION

Thus far, the discussions have focused primarily on the basic ATPG algorithms. As circuits have become increasingly larger and more complex, more powerful ATPG algorithms are needed. In particular, the handling of sequential circuits is a must, because not all circuits may have the luxury of having a full-scan inserted. Next, deterministic ATPGs may face tremendous hurdles when dealing with the need to generate a sequence of many vectors. In this regard, simulation-based ATPGs may be better suited. Finally, the stuck-at fault model may be insufficient in capturing defects that occur at the deep-submicron or nanoscale designs. Such defects include delay faults and bridging faults. This section addresses how the basic ATPG can be extended to deal with these issues.

### 14.5.1 Sequential ATPG: Time frame expansion

Test generation for sequential circuits bears much similarity with that for combinational circuits. However, one vector may be insufficient to detect the target fault, because the excitation and propagation conditions may necessitate some of the flip-flop values to be specified at certain values. The general model for a sequential circuit is shown in Figure 14.39, where flip-flops constitute the memory/state elements of the design. All the flip-flops receive the same clock signal, so no multiple clocks are assumed in the circuit model.

Figure 14.40 illustrates an example of a sequential circuit that is unrolled into several time frames, also called an ***iterative logic array*** (ILA) of the

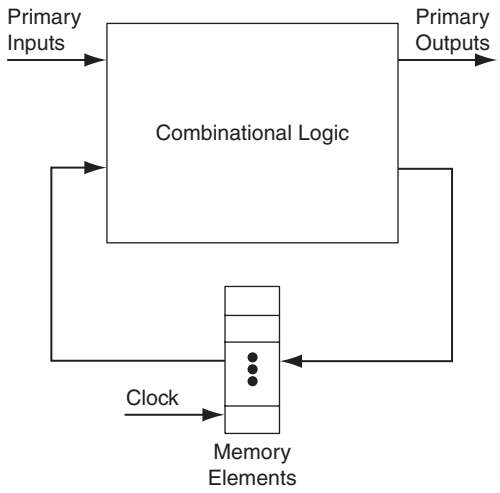



**FIGURE 14.39**

Model of a sequential circuit.

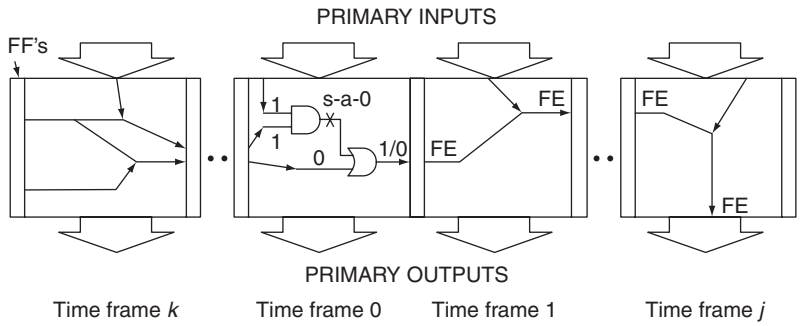



**FIGURE 14.40**

An iterative logic array (ILA) model.

circuit. For each time frame, the flip-flop inputs from the previous time frame are often referred to as **pseudo primary inputs** with respect to that time frame, and the output signals to feed the flip-flops to the next time frame are referred to as **pseudo primary outputs**. Note that in any unrolled circuit, a target fault is present in every time frame.

When the test generation begins, the first time frame is referred to as time frame 0. An ATPG search similar to a combinational circuit is carried out. At the end of the search, a combinational vector is derived, where the input vector consists of primary inputs and pseudo primary inputs. The fault-effect for the target fault may be sensitized to either a primary output of the time frame or a pseudo primary output. If at least one pseudo primary input has been

specified, then the search must attempt to justify the needed flip-flop values in time frame −1. Similarly, if fault-effects only propagate to pseudo primary outputs, the ATPG must try to propagate the fault-effects across time frame + 1. Note that this results in a **test sequence** of vectors. As opposed to combinational circuits, in which a single vector is sufficient to detect a detectable fault, in sequential circuits a test sequence is often needed.

One question naturally arises: Should the ATPG first attempt the fault excitation via several time frames −1, −2, etc., or should the ATPG attempt to propagate the fault-effect through time frames 1, 2, etc.? It can be observed that in propagating the fault-effect in time frame 1, the search may place additional values on the flip-flops between the boundary of time frames 0 and 1. These added constraints propagate backward and may add additional values needed at the pseudo primary inputs at time frame 0. In other words, if the ATPG first justifies the pseudo primary inputs at time frame 0, it would have missed the additional constraints placed by the propagation. Therefore, the ATPG first tries to propagate the fault-effect to a primary output via several time frames, with all the intermediate flip-flop values propagated back to time frame 0. Then, the ATPG proceeds to justify all the pseudo primary input values at time frame 0.

Although easy to understand, the process can be very complex, for example, if the fault-effect has propagated forward for three time frames: time frames 1, 2, and 3. Now in time frame 4, suppose the ATPG successfully propagates the fault-effect to a primary output (*i.e.*, it has derived a vector at time frame 4), it must go back to time frame 3 to make sure the values assigned to the flip-flops at the boundary between time frames 3 and 4 are, indeed, possible. It must perform this check for time frames 2, 1, and 0. If at any time frame a conflict occurs, the vector derived at time frame 4 is actually invalid, because it is not justifiable from the previous vectors. At this time, a backtrack occurs in time frame 4, and the ATPG must try to find a different solution vector #4. This process is repeated.

One way to reduce the complexity discussed is to try to propagate the fault-effect in an unrolled circuit instead of propagating the fault-effect time frame by time frame. In doing so, a k-frame combinational circuit is obtained, say $k = 256$, and the ATPG views the entire 256-frame circuit as one large combinational circuit. However, the ATPG must keep in mind that the target fault is present in all 256 time frames. This eliminates the need to check for state boundary justifiability and allows the ATPG to propagate the fault-effect across multiple time frames at a time.

When the fault-effect has been propagated to at least one primary output, the pseudo primary inputs at time frame 0 must be justified. Again, the justification can be performed in a similar process of viewing an unrolled 256-frame circuit. As before, the ATPG must ensure that the fault is present in every time frame of the unrolled circuit.

**HITEC** [Niermann 1991] is a popular sequential test generator that performs the search similar to the discussed methods with a 9-valued algebra. In addition, it uses the concept of **dominators** to help reduce the search complexity. A dominator for a target fault is a gate in the circuit through which

the fault-effect must traverse [Kirkland 1987]. Therefore, for a given target fault, all inputs of any dominator gate that are not in the fanout cone of the fault must be assigned to noncontrolling values to detect the fault.

The concept of controllability and observability metrics can be extended to sequential circuits such that the backtrace routine would prefer to backtrace toward primary inputs and those easy-to-justify flip-flops. The use of sequential testability metrics allows the ATPG to narrow the search space by favoring the easy-to-reach states and avoiding getting into difficult-to-justify states.

The computational complexity of a sequential ATPG is intuitively higher than that of the combinational ATPG. Therefore, aggressive learning can help to reduce the computational cost. For instance, if a known subset of unreachable states is available, this information can be used to allow the ATPG to backtrack much sooner when an intermediate state is unreachable. This can avoid successive justification of an unreachable state. Likewise, if a justification sequence has been successfully computed for state S before, and a different target fault requires the same state *S*, the previous justification sequence can be used to guide the search. Note that, because the target faults are different, the justification sequence may not simply be copied from the solution for one fault to another.

For large circuits, deterministic ATPGs may suffer from a potentially large number of backtracks. Thus, in the past two decades, effort on **simulation-based ATPGS** has yielded much success, presenting themselves as a viable alternative to deterministic ATPGs. One class of nondeterministic ATPGs is the **genetic algorithm–based** (GA-based) ATPG. There have been numerous GA-based ATPGs proposed over the years. For example, **CONTEST** [Agrawal 1989] targets test generation in three phases, each having its own distinct fitness measure. **GATEST** [Rudnick 1994] distinguishes fault detection from those that only propagate to flip-flop boundaries. **DIGATE** [Hsiao 1996] targets individual faults and uses distinguishing sequences to help propagate the faults from flip-flops to a primary output.

**STRATEGATE** [Hsiao 1997; 2000] addresses fault excitation by justifying the needed state as well. Although GA-based ATPGs have achieved success, the underlying fault simulation engine may incur excessive computational cost. In recent years, approaches that use logic simulation rather than fault simulation have been proposed [Pomeranz 1995; Guo 1999; Giani 2001; Sheng 2002; Wu 2004]. Logic-simulation–based test generators usually target some inherent "property" in the fault-free circuit and try to derive test vectors that exercise these properties. In general, the property used often relates to the states reached by the test sequence.

### 14.5.2 **Delay fault ATPG**

Today's integrated circuits are seeing an escalating clock rate, shrinking dimensions, increasing chip density, etc. Consequently, there arises a class of defects that would affect the functionality of the design if the chip were run at a high speed.

In other words, the design is functionally correct when it is operated at a slow clock. This type of defect is referred to as a **delay defect**. Although the conventional stuck-at testing can catch some delay defects, the stuck-at fault model is insufficient to model delay defects satisfactorily. This has prompted engineers and researchers to propose a variety of methods and fault models for detecting speed failures. Among the fault models are the *transition fault* [Levendel 1986; Waicukauski 1987; Cheng 1993], the *path-delay fault* [Smith 1985], and the *segment delay fault* [Heragu 1996]. The path-delay fault model considers the cumulative effect of the delays along a specific combinational path in the circuit. If the cumulative delay in a faulty circuit exceeds the clock period for the path, then the test pattern that can exercise this path will fail the chip. The segment delay fault model targets path segments instead of complete paths.

Because a transition has to be launched to propagate across a given path, two vectors are needed. The first vector initializes the circuit nodes, and the second vector launches a transition at the start of a path and ensures that the transition is propagated along the given path. Given a path P, a signal is an **on-input** of *P* if it is on *P*. Conversely, a signal is an **off-input** of *P* if it is an input to a gate in *P* but is not an on-input of *P*. A path-delay fault can be a rising fault, where a rising transition is at the start of the path, or a falling fault, where a falling transition is at the start of the path. The rising and falling path-delay faults are denoted with the up-arrow $\uparrow$ and the down-arrow $\downarrow$ before path P, respectively. For example, $\uparrow g_1 g_4 g_7$ is a rising path that traverses through gates $g_1$, $g_4$, and $g_7$.

Delay tests can be applied three different ways: **launch-on-capture** (also called broad-side [Savir 1994] or double-capture [Wang 2006]), **launch-on-shift** (also called skewed-load [Savir 1993]), and **enhanced-scan** [Dervisoglu 1991]. In launch-on–capture-based testing, the first n-bit vector is scanned into the circuit with $n$ scan flip-flops at a slow speed, followed by another clock that creates the transition. Finally, an at-speed functional clock is applied that captures the response. Thus, only one vector has to be stored per test, and the second vector is directly derived from the initial vector by pulsing the clock. In launch-on–shift-based testing, the first $n - 1$ bits of an n-bit vector are shifted in at a slow speed. The final nth shift is performed, and it is also used to launch the transition. This is followed by an at-speed quick capture. Similar to launch-on-capture, only one vector has to be stored per test, because the second vector is simply the shifted version of the first vector. Finally, in enhanced-scan testing, both vectors in the vector pair ($V_1$, $V_2$) have to be stored in the tester memory. The first vector $V_1$ is loaded into the scan chain, followed by its immediate application to initialize the circuit under test. Next, the second vector is scanned in, followed by an immediate application and capture of the response. Note that the node values in the circuit are preserved during the shifting-in of the second vector $V_2$. To achieve this, a **hold-scan design** [Dervisoglu 1991] is required.

Because both launch-on-capture and launch-on-shift place constraints on what the second vector can be, they will achieve lower fault coverage compared with enhanced-scan. However, enhanced-scan comes at a price of

**hold-scan cells** (enhanced-scan cells [Wang 2006]), which consume more chip area. This may not be viewed as a huge negative in microprocessors and some custom-designed circuits, because hold-scan cells are used to prevent the combinational logic from seeing the values being shifted. This is done because the intermediate state of the scan cells may cause contention in some of the signals in the logic, as well as reducing the power consumption in the combinational logic during the shifting of the data in scan cells. In addition, hold-scan cells also help increase the diagnostic capability on failing chips in which the data captured in the scan chain can be retrieved.

In terms of test data volume, enhanced-scan tests may actually require less storage to achieve the same delay fault coverage. In other words, for launch-on-capture or launch-on-shift to achieve the same level of fault coverage, many more patterns may have to be applied.

Unlike stuck-at faults, where a fault is either detected or not detected by a given test vector, a path-delay fault may be detected by different test patterns (consisting of two vectors) with differing levels of quality. In other words, some test patterns can detect a path-delay fault only with certain restrictions in place. Higher quality test patterns place more restrictions on sensitization of the path. On the other hand, similar to stuck-at faults, some paths may be untestable if the sensitization requirement for a given path is not satisfiable.

For designs with two interactive clock domains, modifications can be made to allow for tests. For example, the following at-speed delay test approaches can be used for both launch-on-capture and launch-on-shift architectures: **one-hot double-capture, aligned double-capture**, and **staggered double-capture** [Bhawmik 1997; Wang 2006, 2007b].

If tests were possible for all the paths in a circuit, we would not need any additional test vectors for capturing the delay defects. However, because very few paths are robustly testable, and the number of path-delay faults is exponential to the number of circuit lines, other delay fault models have been proposed. For example, transition tests have been generated to improve the detection of speed failures in microprocessors [Tendulkar 2002], as well as ***application-specific integrated circuits*** (ASICs) [Hsu 2001]. These reasons make transition faults popular in industry.

Similar to the stuck-at fault model, two transition faults are possible at each node of the circuit: *slow-to-rise* and *slow-to-fall*. A test pattern for a transition fault consists of a pair of vectors ($V_1$, $V_2$), where $V_1$ (called the *initial vector*) is required to set the target node to an initial value and $V_2$ (called the *test vector*) is required to launch the corresponding transition at the target node and also propagate the fault effect to a primary output [Waicukauski 1987; Savir 1993].

Transition tests can also be applied in three different ways as for the other delay fault models discussed earlier: launch-on-capture, launch-on-shift, and enhanced scan. As with path-delay tests, because both launch-on-capture and launch-on-shift place constraints on what the second vector can be, they will achieve lower transition fault coverage compared with enhanced-scan.

### 14.5.3 Bridging fault ATPG

Recall that bridging faults are those faults that involve a short between two signals in the circuit. Given a circuit with $n$ signals, there are potentially $n \times (n - 1)$ possible bridging faults. However, practically, only those signals that are locally close on the die are more likely to be bridged. Therefore, the total number of bridging faults can be reduced to be linear in the number of signals in the circuit.

Consider two signals $x$ and $y$ in the circuit that are bridged. This bridging fault will not be excited unless different values are placed on $x$ and $y$. Note that the actual voltage at $x$ and $y$ may be different because of the resistance value of the bridge. Subsequently, the logic that takes $x$ as its input may interpret the logic value differently from the logic that takes $y$ as its input. To reduce the complexity, five common bridging fault models are often used:

1. AND bridge—The faulty value of the bridge for $x'$ and $y'$ is taken to be the logical AND of $x$ and $y$ in the original fault-free circuit.
2. OR bridge—The faulty value of the bridge for $x'$ and $y'$ is taken to be the logical OR of $x$ and $y$ in the original fault-free circuit.
3. $x$ DOM y bridge—$x$ dominates $y$; in other words, the faulty value of the bridge for both $x'$ and $y'$ is taken to be the logic value of $x$ in the fault-free circuit.
4. $x$ DOM1 $y$ bridge—$x$ dominates $y$ if $x = 1$; in other words, the faulty value of $x'$ is unaffected, but the faulty value for $y'$ is taken to be the logical OR of $x$ and $y$ in the fault-free circuit.
5. $x$ DOM0 $y$ bridge—x dominates y if $x = 0$; in other words, the faulty value of $x'$ is unaffected, but the faulty value for $y'$ is taken to be the logical AND of $x$ and y in the fault-free circuit.

Figure 14.41 illustrates the faulty circuit models corresponding to each of these five bridge types. If a path exists between x and y, then the bridging fault is said to be a **feedback-bridging fault**. Otherwise, it is a **non-feedback-bridging fault**. Figure 14.42 illustrates a feedback-bridging fault. In this figure, if $abc = 110$, then in the fault-free circuit $z = 0$. If the bridge is an AND-bridge, then a cycle would result. In other words, a becomes 0 and in turn makes z = 1. Because $a = 1$ initially, it will again try to drive $z = 0$, resulting in an infinite loop around the bridge. For the following discussion, only non-feedback bridging faults will be considered.

Testing for bridging faults is similar to a constrained stuck-at ATPG. In other words, when testing for the AND-bridge($x$, $y$), either (1) $x$/0 has to be detected with $y = 0$ or (2) $y$/0 has to be detected with $x = 0$ [Williams 1973]. A conventional stuck-at ATPG can be modified to handle the added constraint. Likewise, the ATPG can be modified for other bridging fault types.

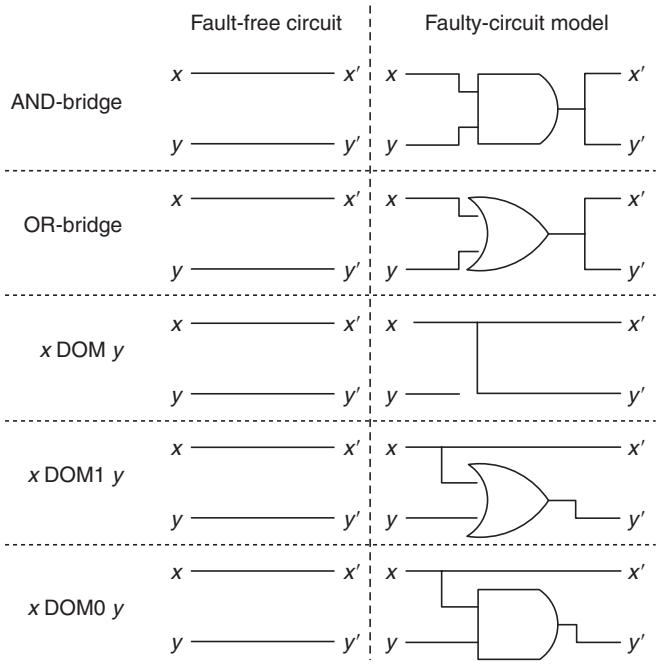



**FIGURE 14.41**

Bridging fault models.

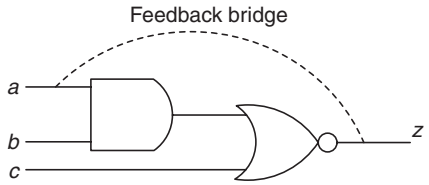



**FIGURE 14.42**

A feedback bridging fault.

## 14.6 CONCLUDING REMARKS

For fault simulation, both event-driven simulation and compiled-code simulation techniques can be found in commercially available ***electronic design automation*** (EDA) applications. The fault simulators can be stand-alone tools or used as an integrated feature in the ATPG programs. As a stand-alone tool, concurrent fault simulation with the event-driven simulation technique is used in Verifault-XL (from Cadence Design Systems [Cadence 2008]) and TurboFault (from SynTest Technologies [SynTest 2008]). As an integrated feature in ATPG, bitwise parallel simulation with the compiled-code simulation technique is widely used

in modern commercial ATPG programs, including Encounter Test (from Cadence Design Systems), FastScan (from Mentor Graphics [Mentor 2008]), TetraMAX (from Synopsys [Synopsys 2008]), and TurboScan (from SynTest Technologies).

As we move to the nanometer age, we have started to see nanometer designs that contain hundreds of millions of transistors. We anticipate the semiconductor industry will completely adopt the scan method for quality considerations. As a result, it is becoming imperative that advanced techniques for both logic simulation and fault simulation be developed to address the high-performance and high-capacity issues, in particular, for addressing new fault models, such as transition faults [Waicukauski 1986], path-delay faults [Schulz 1989], bridging faults [Li 2003], and small delay defects [Sato 2005; Hamada 2006]. At the same time, more innovations are needed in developing advanced concurrent fault simulation techniques, because at present designs based on the scan method are still not 100% scan testable. Fault simulation with functional patterns is important for at-speed test applications to detect small delay faults and achieve the ***parts-per-million*** (PPM) defect level goals.

The theory and implementation of an ATPG engine have also been described in detail in the second half of this chapter. Several algorithms were laid out with specific examples given. Advanced ATPG algorithms were discussed where sequential ATPG and ATPG for non-stuck-at faults were covered. Test generation remains to be an important research area as circuit sizes and complexities continue to increase. New and powerful algorithms are needed to cope with the increased complexity. In addition, with nanoscale feature sizes, new defect types and hence new fault models will be needed in future ATPGs.

Should there be defective chips that were uncovered by the test set, fault diagnosis and failure analysis are often subsequently performed to identify the causes and further reduce the defect level in the future. To ease the burden of fault diagnosis and failure analysis, adding ***design-for-debug-and-diagnosis*** (DFD), ***design-for-reliability*** (DFR), ***design-for-manufacturability*** (DFM), and ***design-for-yield*** (DFY) features can be implemented in the design. These features and techniques are extensively discussed in [Wang 2006, 2007a]. Finally, successful ATPG algorithms not only can help in the area of manufacturing tests, but they also provide much insight to other EDA problems, such as synthesis and verification.

## 14.7 EXERCISES

**14.1. (Equivalence Fault Collapsing)** How many uncollapsed single stuck-at faults are there in circuit *M* shown in Figure 14.43 Please perform equivalence fault collapsing with the simple_EFC algorithm. How many equivalence collapsed faults do you have?

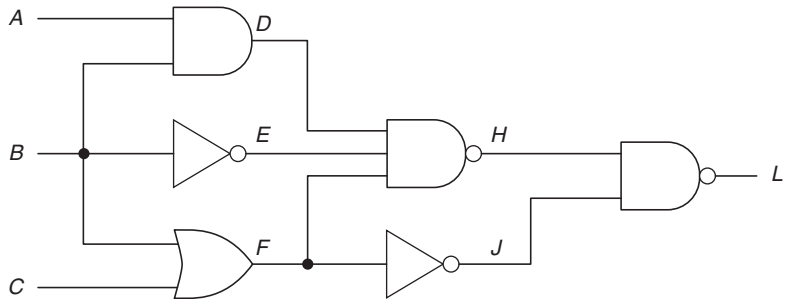



**FIGURE 14.43**

Circuit *M*.

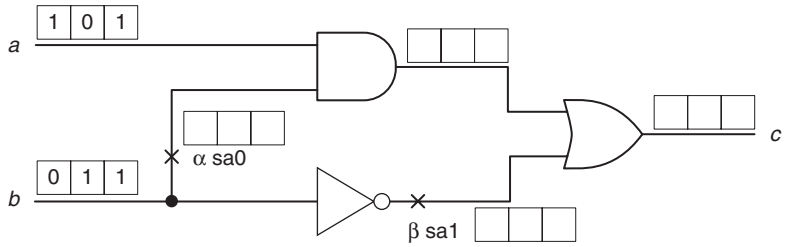



**FIGURE 14.44**

An example circuit *K*.

**14.2. (Dominance Fault Collapsing)** Continued from Exercise 14.1. Please perform dominance fault collapsing with the simple_DFC algorithm. How many dominance collapsed faults do you have?

**14.3. (Dominance Fault Collapsing)** For the circuit in Figure 14.9, please explain why $K/0$ and $K/1$ faults can be removed from the dominance collapsed fault list. Also explain why $F/1$ and $F/0$ can be removed.

**14.4. (Parallel-Pattern Single-Fault Propagation)** For circuit $K$ shown in Figure 14.44 and two given stuck-at faults shown in Figure 14.44, use the parallel-pattern single-fault propagation fault simulation technique to identify which faults can be detected by the given test patterns.

**14.5. (Parallel Fault Simulation)** Repeat Exercise 14.4 by use of parallel fault simulation.

**14.6. (Concurrent Fault Simulation)** Repeat Exercise 14.5 with concurrent fault simulation.

**14.7. (Random Test Generation)** Given a circuit with three primary outputs, $x, y$, and $z$, the fanin cone of $x$ is $\{a, b, c\}$, the fanin cone of y is $\{c, d, e, f\}$, and the fanin cone of z is $\{e, f, g\}$. Devise a pseudo-exhaustive test set for this circuit. Is this test set the minimal pseudo-exhaustive test set?

**14.8. (Random Test Generation)** With the circuit shown in Figure 14.28, compute the detection probabilities for each of the following faults:

a. $e/0$
b. $e/1$
c. $c/0$

**14.9. (Boolean Difference)** With the circuit shown in Figure 14.28, compute the set of all vectors that can detect each of the following faults using Boolean difference:

a. $e/0$
b. $e/1$
c. $c/0$

**14.10. (Boolean Difference)** Assume a single-output combinational circuit, where the output is denoted as $f$. If two faults, $a$ and $b$, are indistinguishable, it means that there does not exist a vector that can detect only one and not the other. Show that $f_a \oplus f_b = 0$ if they are indistinguishable.

**14.11. (*D* Algorithm)** Construct the table for the XNOR operation for the 5-valued logic similar to Tables 14.6, 14.7, and 14.8.

**14.12. (*D* Algorithm)** Consider a three-input AND gate $g$. Suppose $g$ is a $D$-frontier. What are all the possible value combinations the three inputs of $g$ can take such that g is a valid $D$-frontier?

**14.13. (PODEM)** With the circuit shown in Figure 14.28, compute a test vector that can detect each of the following faults by use of PODEM:

a. $e/0$
b. $e/1$
c. $c/0$

**14.14. (FAN)** Consider the circuit shown in Figure 14.37. Suppose the constraint that $y = 1 \rightarrow x = 0$ is given. How could one use this knowledge to reduce the search space when trying to generate vectors in the circuit? For example, suppose the target fault is $y/0$.

**14.15. (Sequential ATPG)** Consider the circuit shown in Figure 14.45. The target fault is $a/0$.

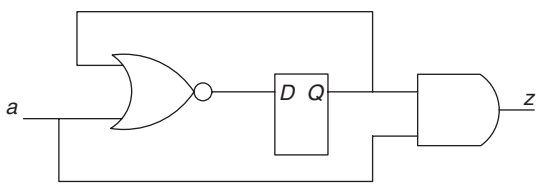



**FIGURE 14.45**

Example sequential circuit.

a. Generate a test sequence for the target fault by use of only 5-valued logic.
b. Generate a test sequence for the target fault by use of 9-valued logic.

**14.16. (Sequential ATPG)** Given a sequential circuit, is it possible that two stuck-at faults, $a/0$ and $a/1$, are both detected by the same vector $v_i$ in a test sequence $v_0, v_1, \ldots, v_k$?

**14.17. (Sequential ATPG)** Consider an **iterative logic array** (ILA) expansion of a sequential circuit, where the initial pseudo primary inputs are fully controllable. Show that the states reachable in successive time frames of the ILA shrink monotonically.

**14.18. (Bridging Faults)** Consider a bridging fault between the outputs of an AND gate $x = ab$ and an OR gate $y = c + d$. What values to $abcd$ would induce the largest current in the bridge?

## ACKNOWLEDGMENTS

We thank Professor Hank Walker of the University of A&M for contributing a portion of the Fault Simulation section; and Professor Xiaoqing Wen of Kyushu Institute of Technology and Professor Charles E. Stroud of Auburn University for reviewing the text and providing helpful comments.

## REFERENCES

### R14.0 Books

[Abramovici 1994] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design, Revised Printing,* IEEE Press, Piscataway, NJ, 1994.

[Bushnell 2000] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI circuits,* Springer Science, New York, 2000.

[Holland 1975] J. H. Holland, *Adaptation in Natural and Artificial Systems,* University of Michigan Press, Ann Arbor, MI, 1975.

[Jha 2003] N. Jha and S. Gupta, *Testing of Digital Systems,* Cambridge University Press, London, 2003.

[Wang 2006] L.-T. Wang, C.-W. Wu, and X. Wen, editors, *VLSI Test Principles and Architectures: Design for Testability,* Morgan Kaufmann, San Francisco, 2006.

[Wang 2007a] L.-T. Wang, C. E. Stroud, and N. A. Touba, editors, *System-on-Chip Test Architectures: Nanometer Design for Testability,* Morgan Kaufmann, San Francisco, November 2007.

### R14.1 Fault Collapsing

[McCluskey 1971] E. J. McCluskey and F. W. Clegg, Fault equivalence in combinational logic networks, *IEEE Trans. on Computers*, C-20(11), pp. 1286–1293, November 1971.

## R14.2 Fault Simulation

[Abramovici 1984] M. Abramovici, P. R. Menon, and D. T. Miller, Critical path tracing: An alternative to fault simulation, *IEEE Design & Test of Computers*, 1(1), pp. 83–93, February 1984.

[Butler 1974] T. T. Butler, T. G. Hallin, J. J. Kulzer, and K. W. Johnson, LAMP: Application to switching system development, *Bell System Technical J.*, 53, pp. 1535–1555, October 1974.

[Cheng 1989] W. T. Cheng and M. L. Yu, Differential fault simulation: A fast method using minimal memory, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 424–428, June 1989.

[Goel 1980] P. Goel, Test generation cost analysis and projections, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 77–84, June 1980.

[Jain 1985] S. K. Jain and V. D. Agrawal, Statistical fault analysis, *IEEE Design & Test of Computers*, 2(1), pp. 38–44, February 1985.

[Niermann 1992] T. M. Niermann, W.-T. Cheng, and J. H. Patel, PROOFS: A fast, memory-efficient sequential circuit fault simulator, *IEEE Trans. on Computer-Aided Design*, 11(2), pp. 198–207, February 1992.

[Schulz 1989] M. Schulz, F. Fink, and K. Fuchs, Parallel pattern fault simulation of path delay faults, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 357–363, June 1989.

[Seshu 1965] S. Sesuh and D. N. Freeman, On improved diagnosis program, *IEEE Trans. on Electronic Computers*, Vol. EC-14(1), pp. 76–79, February 1965.

[Ulrich 1974] E. G. Ulrich and T. Baker, Concurrent simulation of nearly identical digital networks, *IEEE Trans. on Computers*, 7(4), pp. 39–44, April 1974.

[Waicukauski 1985] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom, and T. McCarthy, Fault Simulation for Structured VLSI, in *Proc. VLSI System Design*, 6(12), pp. 20–32, December 1985.

[Waicukauski 1986] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, Transition fault simulation by parallel pattern single fault propagation, in *Proc. IEEE Int. Test Conf.*, pp. 542–549, September 1986.

## R14.3 Test Generation

[Breuer 1971] M. A. Breuer, A random and an algorithmic technique for fault detection test generation for sequential circuits, *IEEE Trans. on Computers*, 20(11), pp. 1364–1370, November 1971.

[Chakradhar 1993] S. T. Chakradhar, V. D. Agrawal, and S. G. Rothweiler, A transitive closure algorithm for test generation, *IEEE Trans. on Computer-Aided Design*, 12(7), pp. 1015–1028, July 1993.

[Cox 1994] H. Cox and J. Rajski, On necessary and non-conflicting assignments in algorithmic test pattern generation, *IEEE Trans. on Computer-Aided Design*, 13(4), pp. 515–530, April 1994.

[David 1976] R. David and G. Blanchet, About random fault detection of combinational networks, *IEEE Trans. on Computers*, C-25(6), pp. 659–664, June 1976.

[Fujiwara 1983] H. Fujiwara and T. Shimono, On the acceleration of test generation algorithms, *IEEE Trans. on Computers*, C-32(12), pp. 1137–1144, December 1983.

[Goel 1981] P. Goel, An implicit enumeration algorithm to generate tests for combinational logic circuits, *IEEE Trans. on Computers*, C-30(3), pp. 215–222, March 1981.

[Hsiao 2002] M. S. Hsiao, Maximizing impossibilities for untestable fault identification, in *Proc. Design, Automation, and Test in Europe Conf.*, pp. 949–953, March 2002.

[Iyer 1996a] M. A. Iyer and M. Abramovici, FIRE: A fault independent combinational redundancy algorithm, *IEEE Trans. VLSI Syst*, 4(2), pp. 295–301, June 1996.

[Iyer 1996b] M. A. Iyer, D. E. Long, and M. Abramovici, Identifying sequential redundancies without search, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 457–462, June 1996.

[Kunz 1993] W. Kunz and D. K. Pradhan, Accelerated dynamic learning for test pattern generation in combinational circuits, *IEEE Trans. on Computer-Aided Design*, 12(5), pp. 684–694, May 1993.

[Kunz 1994] W. Kunz and D. K. Pradhan, Recursive learning: A new implication technique for efficient solutions to CAD problems—test, verification, and optimization, *IEEE Trans. on Computer-Aided Design*, 13(9), pp. 1149–1158, September 1994.

[Lisanke 1987] R. Lisanke, F. Brglez, A. J. Degeus, and D. Gregory, Testability-driven random test-pattern generation, *IEEE Trans. on Computer-Aided Design*, 6(6), pp. 1082–1087, November 1987.

[Marques-Silva 1999] J. P. Marques-Silva and K. A. Sakallah, GRASP: A search algorithm for propositional satisfiability, *IEEE Trans. on Computers*, 48(5), pp. 506–521, May 1999.

[Muth 1976] P. Muth, A nine-valued circuit model for test generation, *IEEE Trans. on Computers*, C-25(6), pp. 630–636, June 1976.

[Rajski 1990] J. Rajski and H. Cox, A method to calculate necessary assignments in ATPG, in *Proc. IEEE Int. Test Conf.*, pp. 25–34, October 1990.

[Roth 1966] J. P. Roth, Diagnosis of automata failures: A calculus and a method, in *IBM J. Research and Development*, 10(4), pp. 278–291, July 1966.

[Roth 1967] J. P. Roth, W. G. Bouricius, and P. R. Schneider, Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits, *IEEE Trans. on Electron. Comput.*, EC-16(10), pp. 567–579, October 1967.

[Schnurmann 1975] H. D. Schnurmann, E. Lindbloom, and R. G. Carpenter, The weighted random test-pattern generator, *IEEE Trans. on Computers*, 24(7), pp. 695–700, July 1975.

[Schulz 1988] M. H. Schulz, E. Trischler, and T. M. Sarfert, SOCRATES: A highly efficient automatic test pattern generation system, *IEEE Trans. on Computer-Aided Design*, 7(1), pp. 126–137, January 1988.

[Schulz 1989] M. H. Schulz and E. Auth, Improved deterministic test pattern generation with applications to redundancy identification, *IEEE Trans. on Computer-Aided Design*, 8(7), pp. 811–816, July 1989.

[Seshu 1965] S. Seshu and D. N. Freeman, The diagnosis of synchronous sequential switching systems, *IEEE Trans. on Electron. Comput.*, 11, pp. 459–465, August 1962.

[Shedletsky 1977] J. J. Shedletsky, Random testing: Practicality vs. verified effectiveness, in *Proc. IEEE Int. Symp. on Fault-Tolerant Computing*, pp. 175–179, June 1977.

[Syal 2003] M. Syal and M. S. Hsiao, A novel, low-cost algorithm for sequentially untestable fault identification, in *Proc. ACM/IEEE Design, Automation, and Test in Europe Conf.*, pp. 316–321, March 2003.

[Zhao 2001] J. Zhao, J. A. Newquist, and J. H. Patel, A graph traversal based framework for sequential logic implication with an application to C-cycle redundancy identification, in *Proc. IEEE Int. Conf. on VLSI Design*, pp. 163–169, January 2001.

## R14.4 Advanced Test Generation

[Agrawal 1989] V. D. Agrawal, K.-T. Cheng, and P. Agrawal, A directed search method for test generation using a concurrent simulator, *IEEE Trans. on Computer-Aided Design*, 8(2), pp. 131–138, February 1989.

[Bhawmik 1997] S. Bhawmik, Method and Apparatus for Built-In Self-Test with Multiple Clock Circuits, U.S. Patent No. 5,680,543, October 21, 1997.

[Cheng 1993] K.-T. Cheng, S. Devadas, and K. Keutzer, Delay-fault test generation and synthesis for testability under a standard scan design methodology, *IEEE Trans. on Computer-Aided Design*, 12(8), pp. 1217–1231, August 1993.

[Dervisoglu 1991] B. Dervisoglu and G. Stong, Design for testability: Using scanpath techniques for path-delay test and measurement, in *Proc. IEEE Int. Test Conf.*, pp. 365–374, October 1991.

[Giani 2001] A. Giani, S. Sheng, M. S. Hsiao, and V. Agrawal, Efficient spectral techniques for sequential ATPG, in *Proc. IEEE Design, Automation, and Test in Europe Conf.*, pp. 204–208, March 2001.

[Guo 1999] R. Guo, S. M. Reddy, and I. Pomeranz, Proptest: A property based test pattern generator for sequential circuits using test compaction, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 653–659, June 1999.

[Heragu 1996] K. Heragu, J. H. Patel, and V. D. Agrawal, Segment delay faults: A new fault model, in *Proc. IEEE VLSI Test Symp.*, pp. 32–39, April 1996.

[Hsiao 1996] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, Automatic test generation using genetically engineered distinguishing sequences, in *Proc. IEEE VLSI Test Symp.*, pp. 216–223, April 1996.

[Hsiao 1997] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, Sequential circuit test generation using dynamic state traversal, in *Proc. European Design and Test Conf.*, pp. 22–28, February 1997.

[Hsiao 2000] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, Dynamic state traversal for sequential circuit test generation, *ACM Trans. on Design Automation of Electronic Systems*, 5(3), pp. 548–565, July 2000.

[Hsu 2001] F. F. Hsu, K. M. Butler, and J. H. Patel, A case study of the Illinois scan architecture, in *Proc. IEEE Int. Test Conf.*, pp. 538–547, October 2001.

[Kirkland 1987] T. Kirkland and M. R. Mercer, A topological search algorithm for ATPG, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 502–508, June 1987.

[Levendel 1986] Y. Levendel and P. Menon, Transition faults in combinational circuits: Input transition test generation and fault simulation, in *Proc. Fault-Tolerant Computing Symp.*, pp. 278–283, July 1986.

[Niermann 1991] T. M. Niermann and J. H. Patel, HITEC: A test generation package for sequential circuits, in *Proc. European Design Automation Conf.*, pp. 214–218, February 1991.

[Pomeranz 1995] 1Pomeranz and S. M. Reddy, LOCSTEP: A logic simulation based test generation procedure, in *Proc. Fault-Tolerant Computing Symp.*, pp. 110–119, June 1995.

[Rudnick 1994] E. M. Rudnick, J. H. Patel, G. S. Greenstein, and T. M. Niermann, Sequential circuit test generation in a genetic algorithm framework, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 698–704, June 1994.

[Savir 1993] J. Savir and S. Patil, Scan-based transition test, *IEEE Trans. on Computer-Aided Design*, 12(8), pp. 1232–1241, August 1993.

[Savir 1994] J. Savir and S. Patil, On broad-side delay test, in *Proc. IEEE VLSI Test Symp.*, pp. 284–290, April 1994.

[Sheng 2002] S. Sheng, K. Takayama, and M. S. Hsiao, Effective safety property checking based on simulation-based ATPG, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 813–818, June 2002.

[Smith 1985] G. L. Smith, Model for delay faults based upon paths, in *Proc. IEEE Int. Test Conf.*, pp. 342–349, October 1985.

[Tendulkar 2002] N. Tendulkar, R. Raina, R. Woltenburg, X. Lin, B. Swanson, and G. Aldrich, Novel techniques for achieving high at-speed transition fault coverage for Motorola's microprocessors based on PowerPC instruction set architecture, in *Proc. IEEE VLSI Test Symp.*, pp. 3–8, April 2002.

[Waicukauski 1987] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, Transition fault simulation, *IEEE Design & Test of Computers*, 4(2), pp. 32–38, April 1987.

[Wang 2007b] L.-T. Wang, P.-C. Hsu, and X. Wen, Multiple-Capture DFT System for Detecting or Locating Crossing Clock-Domain Faults During Scan-Test, U.S. Patent No. 7,260,756, August 21, 2007.

[Williams 1973] M. J. Y. Williams and J. B. Angel, Enhancing testability of large-scale integrated circuits via test points and additional logic, *IEEE Trans. on Computers*, C-22(1), pp. 46–60, January 1973.

[Wu 2004] Q. Wu and M. S. Hsiao, Efficient ATPG for design validation based on partitioned state exploration histories, in *Proc. IEEE VLSI Test Symp.*, pp. 389–394, April 2004.

## R14.5 Concluding Remarks

[Cadence 2008] Cadence Design Systems, http://www.cadence.com, April 2004.

[Hamada 2006] S. Hamada, T. Maeda, A. Takatori, Y. Noduyama, and Y. Sato, Recognition of sensitized longest paths in transition-delay test, in *Proc. IEEE Int. Test Conf.*, Paper 11.1, October 2006.

[Li 2003] Z. Li, X. Lu, W. Qiu, W. Shi, and D. M. H. Walker, A circuit level fault model for resistive bridges, *ACM Trans. on Design Automation of Electronic Systems*, 8(4), pp. 546–559, October 2003.

[Mentor 2008] Mentor Graphics, http://www.mentor.com, 2008.

[Sato 2005] Y. Sato, S. Hamada, T. Maeda, A. Takatori, Y. Nozuyama, and S. Kajihara, Invisible delay quality-SDQM model lights up what could not be seen, in *Proc. IEEE Int. Test Conf.* Paper 47.1, November 2005.

[Synopsys 2008] Synopsys, http://www.synopsys.com, October 2003.

[SynTest 2008] SynTest Technologies, http://www.syntest.com, 2008.

[Waicukauski 1986] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, Transition fault simulation by parallel pattern single fault propagation, in *Proc. IEEE Int. Test Conf.*, pp. 542–549, September 1986.