TALLINN TECHNICAL UNIVERSITY
Faculty of Information Technology

Department of Computer Engineering

Chair of Computer Engineering and Diagnostics

# THEORETICAL BACKGROUND FOR THE APPLET “DESIGN AND TEST OF DIGITAL SYSTEMS ON RT-LEVEL” AND RELATED EXERCISES

TALLINN 2003

# TABLE OF CONTENTS

# 1 PRINCIPLES OF HIGH-LEVEL DESIGN REPRESENTATION

## 1.1 LEVELS OF ABSTRACTION

Within the product definition, design and manufacturing process, each person looks at the product from a slightly different point of view and requires specific information to support his or her work. For this reason, each product, and consequently each design, requires several different representations or views, which differ in the type of information that they emphasize. In addition the same representation often requires different levels of detail in different phases of the design. The three most common types of representation that are used are behavioral, structural, and physical representations.

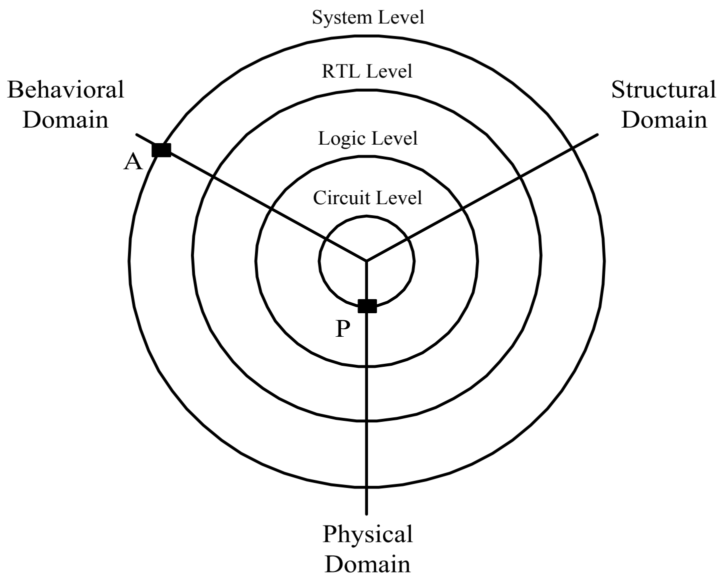


Figure1. The Y-chart

To define and differentiate types of representations, use the Y-chart, a tripartite representation of design, which is shown in Figure 1. The axes in the Y-chart represent three different domains of description: behavioral, structural and physical. Along each axis are different levels of the domain description. As to move farther away from the center of the Y, the level of description becomes more abstract.

Drawing concentric circles on the Y can extend this chart. Each circle intersects the Y-axis at a particular level of representation within a domain. The circle represents all the information known about the design at some point of time. The outer circle is the system level; the next is microarchitectural or register-transfer (RT) level, followed by the logic and circuit levels. Table 1 lists these levels.

In the behavioral domain designers are interested in what a design does, not in how it is built. Usually design is treated as one or more black boxes with a specified set of inputs and outputs and a set of functions describing the behavior of each output in terms of the inputs over time. A behavioral description includes an interface description and description of constraints imposed on a design. The interface description specifies the I/O ports and timing relationships or protocols among signals at those ports. Constraints specify technological relationships that must hold for the design to be verifiable, testable, manufacturable and maintainable.

| Level Name | Behavioral Representation | Structural Representation | Physical Representation |
|---|---|---|---|
| System level | System specification | Blocks | Chip |
| Register-transfer level | RTL-specification | Registers | Macro cells |
| Logic level | Boolean functions | Logic gates | Standard cells |
| Circuit level | Differential equations | Transistors | Contacts |

Table 1. Objects design at different abstraction levels

To describe behavior, transfer functions and timing diagrams are used on a circuit level and Boolean expressions and state diagrams on the logic level. On the RT level, time is divided into intervals called control states or steps. Register-transfer description specifies for each control state the condition to be tested; all registers transfers to be executed, and the next control state to be entered. On the system level variables and language operators are used to express functionality of system components. Variables and data structures are not bound to registers and memories, and operations are not bound to any functional unit or control states. In a system level description, timing is further abstracted to the order in which variable assignments are executed.

A structural representation bridges the behavioral and physical representation. It is one-to-many mapping of a behavioral representation onto a set of components and connections under constraints such as cost, area and delay. The most commonly used levels of structural representation are identified in terms of the basic structural elements used. On the circuit level the basic elements are transistors, resistors and capacitors, while gates and flip-flops are the basic elements on the logic level. ALU's, multipliers, registers; RAM's and ROM's are used to identify register-transfers. Processors, memories and buses are used on the system level.

The physical representation ignores, as much as possible, what the design is supposed to do and binds its structure in space or to silicon. The most commonly used levels in the physical representation are modules, multi-chip modules (MCMs) and printed circuit (PC) boards. [3]

## 1.2 REGISTER-TRANSFER DESIGN

In this section the concept of datapath and control unit, different techniques of specifying control unit and minimizing datapath for the design synthesis on register-transfer level will be presented. Since each RT implementation defines both a control unit and a datapath, we can approach the concept o these parts separately.

### 1.2.1 THE CONCEPT OF DATAPATH

RT-level designs are composed of two interacting parts: datapath and control unit. Usually datapath consists of storage units such as registers, register files, and memories, and combinatorial units such as ALUs, multipliers, shifters etc. Buses connect these units, the input and output ports. The datapath takes the operands from storage units, performs the computation in the combinatorial units, and returns the results to storage units during each state, which is usually equal to one clock cycle. The control unit controls the selection of operands, operations, and the destination for the results by setting proper values of datapath control signals. The datapath also indicates through status signals when a particular value is stored in a particular storage unit or when a particular relation between two data values stored in the datapath is satisfied.

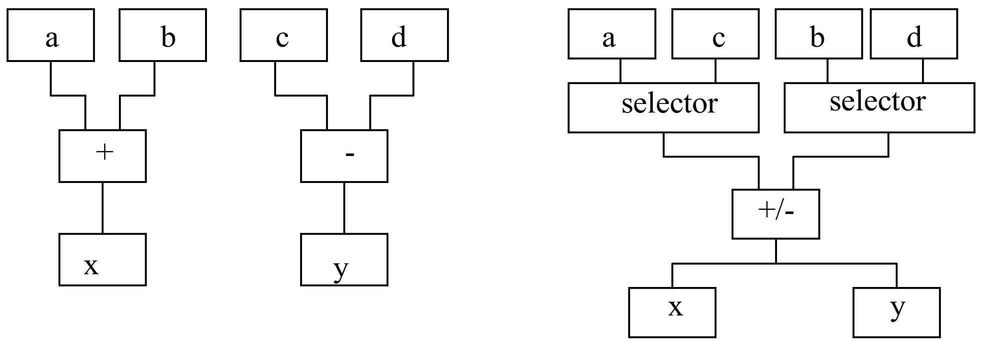
For many high-speed applications simple datapaths are too slow. In order, to increase the performance simple datapath redesign so that several operations could be performed concurrently. These faster datapaths are called *parallel datapath*. The obvious way to parallelize datapath would be to increase the number of registers and use several functional units. But note, that the performance increase in a parallel datapath will depend not just on the number and type of units in the datapath, but also on their connectivity and the amount of parallelism that is available in the algorithm which is executing on datapath. For the best performance/cost ratio, the types of units and their connectivity must match the parallelism in the given algorithm.

There are some general techniques to optimize, to minimize the implementation of the datapath, which based on the following component types used in the datapath: storage components and functional units.

By focusing on the storage components, for example, we note that the variables in the datapath must be stored in registers, register files, and memories. However, since not variables are used at the same time, it is possible for certain variables to share the same register or the same location in a register file or memory. So we can merge the datapath variables in a way that reduces the number of storage locations in the datapath.

Alternatively, certain optimization techniques can focus on minimizing the number of functional units in the datapath. In each state, selected variables are to be assigned new values through various arithmetic, logic or shift operations, each of which can be performed by a separate functional unit. However, since most of these operations are executed in different states, they could share the same functional unit. So, we can reduce the number of units in the datapath by combining different operations into groups, allowing each group of operations to be executed in a single functional unit.

Minimizing the number of functional units in a datapath we deal with functional unit sharing. Functional unit sharing is possible, because within any given state, a datapath will not perform every operation. Therefore, similar operators can be grouped into a single multifunction unit, which will be used more frequently, thus increasing the unit utilization. In some cases, of course, grouping operations in this manner may not reduce the cost of the datapath, since dissimilar operators often require structurally different designs, grouping them can sometimes result in no gain or even in a higher cost.



(a) No shared design (b) Shared design

Figure 2. Functional unit sharing

In this example is assumed that the datapath will perform two different operations, addition and subtraction, on different operands in different states. If we use single-function units, we will get the design shown in Figure 2(a), in which the datapath requires both an adder and a subtractor. We can, however, obtain the same functionality by using only one adder/subtractor and two selectors, as shown in Figure 2(b). Obviously, the second design will be preferable when the cost of an adder/subtractor and two selectors is less than the cost of a separate adder and subtractor.

### 1.2.2 THE CONCEPT OF CONTROL UNIT

Similar to the datapath, a control unit has a set of input and a set of output signals. Each signal is a Boolean variable that can take value of 0 or 1. There are two types of input signals: external signals and status signals. External signals represent the conditions in the external environment on which circuit must respond. The status signals represent the state of the datapath. Their value is obtained by comparing values of selected variables stored in the datapath. There are also two types of output signals: external signals and datapath control signals. External signals identify to the environment that circuit has reached a certain state or finished a particular computation. The datapath controls select the operation for each component in the datapath.

In order to specify control unit we will look at two different techniques. The first technique is based on finite state machines that are usually represented graphically. The second technique, called microprogrammed control, uses a programming representation for control unit.

The first technique we use to specify a control unit is finite state machine. The finite state machine (FSM) consists of a set of states and directions on how to change states. The FSM can be defined abstractly as the quintuple <S, I, O, f, h>, where S, I, and O represent a set of outputs, respectively, and f and h represent the next state and output functions. The next state function f is defined abstractly as a mapping $S * I \rightarrow S$. The FSM model assumes that the time is divided into uniform intervals and these transitions from one state to another occur only at the beginning of each time interval. Therefore, the next state function f defines what the state of the FSM will be in the next time interval given the state and input values in the present interval. The output function h determines the output values in the present state.

There are two different types of FSM, which correspond to two different definitions of the output function *h*. One type is a state-based or Moore FSM, for which *h* is defined as a mapping $S \rightarrow O$. The output symbol is assigned to each state of the FSM, and depends only on the current state. The other type is an input-based or Mealy FSM, for which *h* is defined as the mapping $S * I \rightarrow O$. In this case, a pair of state and input symbols defines an output symbol in each state.

As it was told earlier, the second technique of specifying the control unit is microprogrammed control, which is based on follows:

- A set of control signals that must be asserted in a given state is defined by microinstructions selected in microprogram.
- In order to control different transitions between the states of microprogram are defined by a set of datapath status signals. Which, usually, take a value of 0 or 1.
- Designing a control unit as an algorithm that implements the simpler microinstructions is called microprogramming. The key idea of microprogramming is to represent the asserted values of control lines symbolically, so the microprogramm is a representation of microinstructions.
- Microprogram is a representation of the control unit that will be translated by algorithm to control logic.[6]

# 2 INTRODUCTION TO TESTING

## 2.1 TESTING

*Testing* of a system is an experiment in which the system is exercised and its resulting response is analyzed to ascertain whether it behaved correctly. If incorrect behavior is detected, a second goal of a testing experiment may be to diagnose, or locate, the cause the misbehavior. Diagnosis assumes knowledge of the internal structure of the system under test. These concepts of testing and diagnosis have a broad applicability; consider, for example, medical tests and diagnoses or debugging a computer program.

An important problem in testing is *test evaluation*, which refers to determining the effectiveness, or quality of a test. Test evaluation is usually done in the context of a fault model, and the quality of a test, *fault coverage*. Test evaluation is carried out via a simulated testing experiment called *fault simulation*. Fault coverage and fault simulation will be described in this section as well.

## 2.2 TEST GENERATION

*Test generation* (TG) is the process of determining the stimuli necessary to test a digital system. TG depends primarily on the testing method employed. On-line testing methods do not require TG. Little TG effort is needed, when a feedback shift register working as a pseudorandom sequence generator provides the input patterns. Automatic TG (ATG) refers to TG algorithm that, given a model of a system, can generate tests for it. Random TG (RTG) is a simple process that involves only generation of random vectors. However to achieve a high-quality test a large set of random vectors is needed.

TG can be fault oriented or functional oriented. In fault-oriented TG, ones try to generate tests that will detect (and possibly locate) specific faults. In function-oriented TG, one tries to generate a test that, if it passes, shows that the system performs its specified function.

## 2.3 BRIEF DESCRIPTION OF FAULT MODELS.

Failure modes are manifested on the logical level as incorrect signal values. A fault is a model that represents the effect of a failure by means of the change that is produced in the system signal. Several defects are usually mapped to one fault model. But some defects can be also represented by more than one fault model. The table 1 lists the common fault models.

| **Fault model** | **Description** |
|---|---|
| Single stuck-at faults (SSF) | One line takes the value 0 or 1. |
| Multiple stuck-at faults | Two or more lines have fixed values, not necessarily the same. |
| Bridging faults | Two or more lines that are normally independent become electrically connected. |
| Delay faults | A fault is caused by delays in one or more paths in the |

| | |
|---|---|
| | circuit. |
| Intermittent faults | Caused by internal parameter degradation. Incorrect signal values occur for some but not all states of the circuit. Degradation is progressive until permanent failure occurs. |
| Transient faults | Incorrect signal values caused by coupled disturbances. Coupling may be via power bus capacity or inductive coupling. |

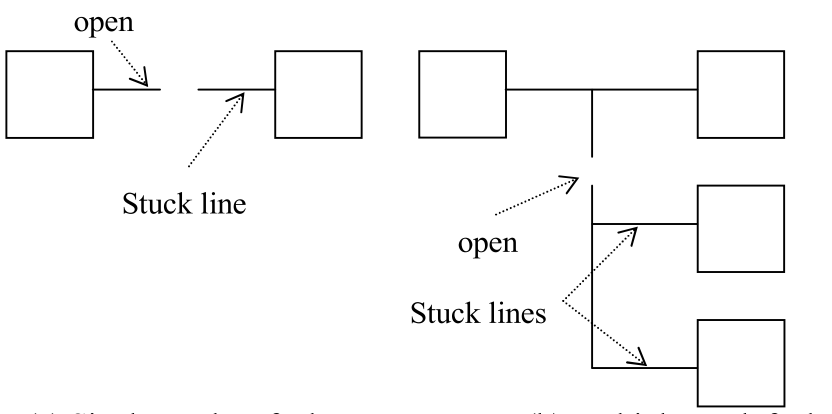
Table 2. Most commonly used Fault Models.

Faults defined in conjunction with a structural model are referred to as structural faults; their effect is to modify the interconnections among components. Functional faults are defined in conjunction with a functional model.

Although intermittent and transient faults occur often, their modeling requires statical data on their probability of occurrence. These data are needed to determine how many times an off-line testing experiment should be repeated to maximize the probability of detecting a fault that is only sometimes present in the circuit under test. Unfortunately, this type of data is usually not available. Intermittent and transient faults are better dealt with by on-line testing.

The simplifying single-fault assumption is justified by the frequent testing strategy, which states that we should test a system often enough so that the probability of more than one fault developing between two consecutive testing experiments is sufficiently small. Thus if maintenance intervals for a working system are too long, we are likely to encounter multiple faults. But even when multiple faults are present, the tests derived under a single-fault assumption are usually applicable for the detection of multiple faults, because, in most cases, a multiple fault can be detected by the tests designed for the individual single faults that compose the multiple one.

In general, structural fault models assume that components are fault-free and only their interconnections are affected. Typical faults affecting interconnections are shorts and opens. A short is formed by connecting points not intended to be connected, while an open results from the breaking of a connection. For example, in many technologies, a short between ground or power and a signal line can make the signal remain at a fixed voltage level. The corresponding logical fault consists of the signal being stuck at a fixed logic value $v$ ($v \in \{0, 1\}$), and denoted by *s-a-v*. A short between two signal lines usually creates a new logic function. The logical fault representing such a short is referred to as a bridging fault. The effect of an open on undirectional signal line with only one fanout is to make the input that has become unconnected due to the open assume a constant logic value and hence appear as a stuck fault (Figure 3(a)). An open in a signal line with fanout may result in multiple stuck fault, as it shown in Figure 3(b).



(a) Single stuck at fault (b) Multiple stuck fault
Figure 3. Stuck faults caused by opens

## 2.4 SINGLE STUCK –FAULT MODEL

The single-stuck fault model is also referred to as the classical or standard fault model because it has been the first and the most widely studied and used. Although its validity is not universal, its usefulness results from the following attributes:

- It represents many different physical faults.
- It is independent of technology, as the concept of a signal line being stuck at a logic value can be applied to any structural model
- Compared to other fault models, the number of SSFs in a circuit is small.
- SSFs can be used to model other types of faults.

The last point is illustrated in Figure 4. To model a fault that changes the behavior of the signal line z, we add to original circuit a multiplexor that realizes the function

$$z' = z \text{ if } f = 0$$
$$z' = z_f \text{ if } f = 1$$

The new circuit operates identically to the original circuit for f=0 and can realize any faulty function $z_f$ by inserting the fault *s-a-1*. For example connecting $x$ to $z_f$ would create the effect of a functional fault that changes the function of the inverter from $z = \overline{x}$ to $z = x$. Connecting $x$ to $z_f$ via an inverter with a different delay would create the effect of delay model.

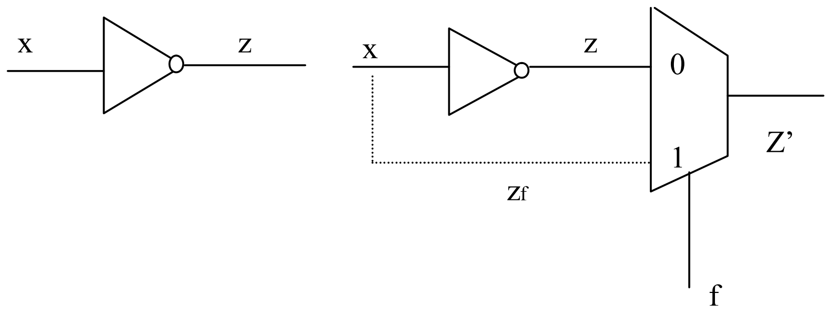


Figure 4. Model modification

## 2.5 MULTIPLE STUCK-FAULT MODEL

The multiple stuck-fault (MSF) models are a straightforward extension of the SSF model in which several lines can be simultaneously stuck. If we denote by n number of possible SSF sites, there are *2n* possible SSFs, but there are *$(3^n - 1)$* possible MSFs (which include the SSFs). If we assume that the multiplicity of a fault, the number of lines simultaneously stuck, is no greater than a constant *k,* then the number of possible MSFs is

$$\sum_{i=1}^{k} \binom{n}{i} 2^i$$

This is usually too large number to allow dealing with all multiple faults. To detecting MSFs, it is always possible to use exhaustive and pseudoexhaustive testing. However, it is not practical for large circuits. The most important factors that affect the detectability of MSFs are the number of primary outputs and reconverging fanouts.

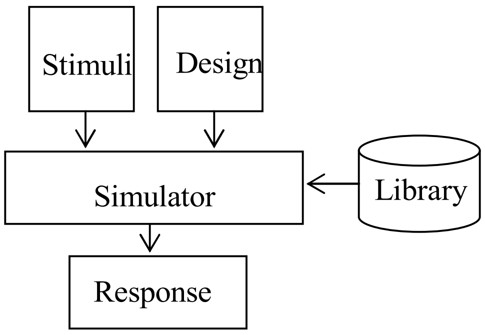


Figure 5. Elements of simulation

We are not going to describe other fault models, such as bridging faults, delay faults and temporary faults, the short description of these models is given in Table 2.

## 2.6 FAULT SIMULATION

Fault simulation is performed during the design cycle to achieve the following goals:

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

To perform the task of fault simulation, the fault simulation program requires, in addition to the circuit model, the stimuli, and the responses of a good circuit to the stimuli, a fault model and a fault list.

As it was mentioned above, there are different fault models, and the most widely used is the stuck-at model. The responses deduced by the fault simulator are used to determine the fault coverage.

The fault simulation process is shown in Figure 5. A fault is considered from the list and a pattern is applied to the circuit. If the fault is detected, it is dropped from the fault list and the next fault is considered. Otherwise, another pattern is applied; the fault is then considered undetectable by the test and is removed from the fault list. The process is continued until the fault list is empty.

After fault simulation, the faults are either detected or undetected. The fault is detected if it has been controllable and observable by one of the patterns in the test set. In such case, at least one of the primary outputs of the faulty circuit is different from the good circuit. Otherwise, it is not detected by any of the patterns of the test set.

## 2.7 FAULT COVERAGE

The effectiveness of the test set is quantifiable. It is the percentage of the faults detected by a test and is known as fault coverage, defined as

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

And a more realistic expression is

$$\text{fault coverage} = \frac{\text{faults detected}}{\text{detectable faults}}$$

In other words, this is a percentage of detectable faults in the circuit under test (CUT) that are detected by a test set. The set is complete if its fault coverage is 100%. The level of fault coverage is desirable but rarely attainable in most practical circuits. Moreover, the 100% fault coverage does not guarantee that the circuit is fault-free. The test checks only for failures that can be represented by the model used, such as stuck-at-fault-model. Other failures are not necessarily detected.

## 2.8 DETERMINISTIC TEST

### 2.8.1 INTRODUCTION TO DETERMINISTIC TEST

In contrast to Random Test Generation, which is generally works without taking into account the function or the structure of the CUT, deterministic test generation produces tests by processing a model on the circuit. But deterministic test is more expensive, but it produces shorter and higher-quality tests.

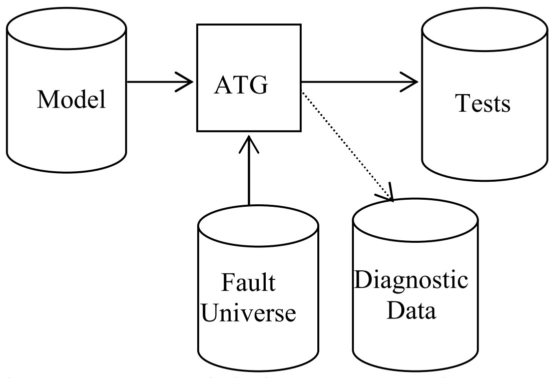


Figure 6. Deterministic test generation system

Deterministic test can be fault-oriented or fault-independent. In a fault-oriented process, tests are generated for specified faults, whereas a fault-independent test works without targeting faults.

Figure 6 shows a general view of a deterministic test generation system. Tests are generated based on a model of the circuit and a given fault model. The generated tests include both the stimuli to be applied and the expected response of the fault-free circuit. Some test generation systems also produce diagnostic data to be used for fault location.
As it was told generated tests based on a given fault model, the fault model in our case is the stuck-at fault model. Using deterministic test generation some heuristic algorithms can be proposed. The best known are the D-algorithm, critical path algorithm, PODEM and SOCRATES.

All algorithms are based on the four main operations processes of excitation, sensitization, justification, and implication. The D-algorithm starts at the faulty line, and its main difficulty is in reconverging fan-out. The critical path algorithm starts from the primary outputs of the circuit and generates a test pattern for several faults, while PODEM starts from the primary inputs.

## 2.8.2 BASIC OPERATIONS OF DETERMINISTIC TEST

To generate the pattern for a stuck –at fault on a line, we need to provoke or excite the fault, sensitize the results to a primary output, and justify the logic values required on the other lines in the circuit. It is needed to find implications of these values on other gates.

To provoke or excite a line is to control it to a logic value that is the complement of the value at which it is stuck; this is equivalent to placing the faulty signal on the line. The signal is a discrepancy from the fault-free circuit. For example, to provoke the stuck-at 1 fault on line W, W/1, of the circuit in Figure 7, we must put 0 on this line, W=0.

It is necessary to sensitize or propagate the fault to a primary output in order to observe it. The path from the faulty location to the primary output is a sensitizing or propagation path. A fault may have more than one sensitizing path to the same output or to different outputs. The fault W/1 has one sensitizing path: trough G3, G4, and G6. To sensitize the fault to the output of G3, we must have E=1. Finally to propagate the fault to the primary output, Z, we need to have H=1. The values on E and H need to be justified to the primary inputs.

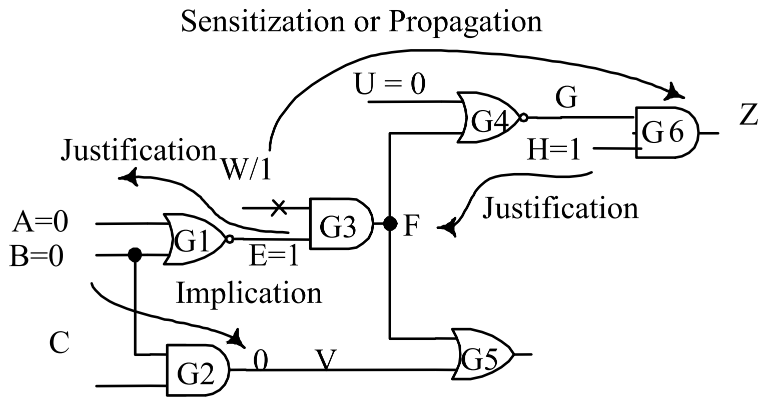


Figure 7. Test pattern generation terminology

We justify 1 on E by having A =B =0. Next we find the implication of B on gate G2. Sometimes in propagating and justifying we encounter a conflict because some of the lines we need to control have values already assigned. In such cases it is said that we encountered an inconsistency. [2]

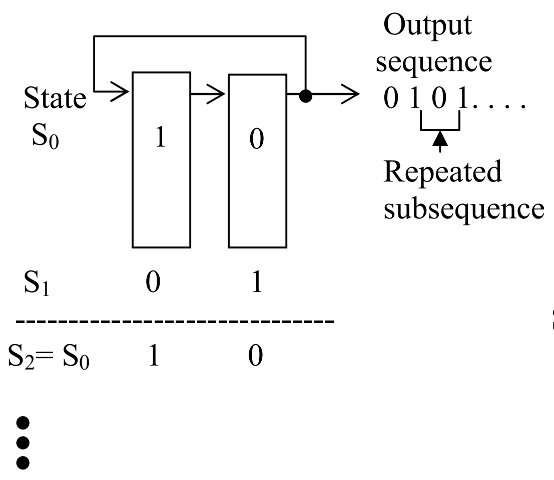
# 3 THEORY AND OPERATION OF LINEAR FEEDBACK SHIFT REGISTER

In this section some of the formal properties associated with linear feedback shift registers will be presented. LFSRs are used extensively in two capacities in DFT and BIST designs, as a source of pseudorandom binary test sequences and as a means to carry out response compression – known as signature analysis.

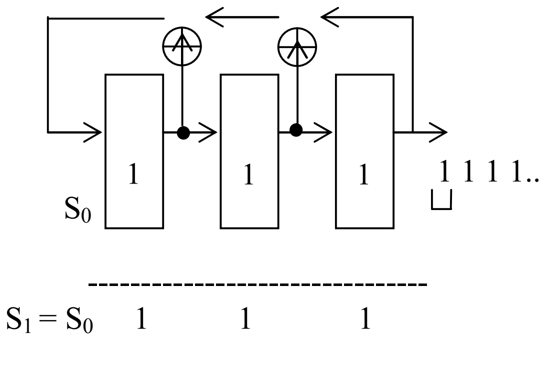
## 3.1 BRIEF DESCRIPTION OF LFSRs STRUCTURE

Consider the feedback shift registers shown in Figure 8. These circuits are all autonomous – they have no inputs except for clocks. Each cell is assumed to be a clocked D flip-flop. It is well known that such circuits are cyclic in the sense that they clocked repeatedly; they go through a fixed sequence of states. For example, a binary counter consisting of *n* flip-flops would go through a fixed sequence of states 0, 1, $2^{n-1}$, 0, 1,…. The maximum number of states for such a device is $2^n$. The shift register shown in a Figure 8(a) cycles through only two states. If the initial state were 00 or 11, it would never change state. An *n*-bit shift register cycles through at most *n* states. Notice that the output sequence generated by such a device is also cyclic. The circuit of Figure 8(b) starting in the initial state 111 (or 000) produces a cyclic sequence of states of length 1. The sequence generated for the circuit of Figure 8(c) if the initial state is 011 is shown in Figure 8(b).

In Figure 8(d) is illustrated the case where sequence generated by the feedback shift register is of the length $(2^3-1)$. The circuit of Figure 8(d) is said to be a maximal-length shift register, since it generated a cyclic state sequence of length $(2^n-1)$, as long as its initial state is not all zeros. Also if one of these circuits generates a cyclic state sequence of length *k*, then the output sequence also repeats itself every *k* clock cycles.



(a)

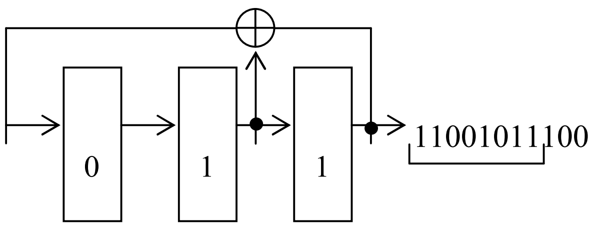


(b)

| | | | |
|---|---|---|---|
| $S_0$ | 0 | 1 | 1 |
| $S_1$ | 0 | 0 | 1 |
| $S_2$ | 1 | 0 | 0 |
| $S_3$ | 1 | 1 | 0 |
| $S_4 = S_0$ | 0 | 1 | 1 |

●
●
●

(c)



| | | | |
|---|---|---|---|
| $S_1$ | 0 | 0 | 1 |
| $S_2$ | 1 | 0 | 0 |
| $S_3$ | 0 | 1 | 0 |
| $S_4$ | 1 | 0 | 1 |
| $S_5$ | 1 | 1 | 0 |
| $S_6$ | 1 | 1 | 1 |
| $S_7 = S_0$ | 0 | 1 | 1 |

(d)

Figure 8. Feedback shift registers

A *linear circuit* is a logic network constructed from the following basic components:

- unit delays or D flip-flops;
- modulo- 2 adders;
- modulo- 2 scalar multipliers;

In the analysis of such circuits, all operations are done modulo 2. The truth table for modulo- 2 addition and subtraction is shown below.

| $\pm$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

Thus $x + x= -x - x= x - x= o$

Such a circuit is considered to be a linear since it preserves the principle of superposition, because its response to a linear combination of stimuli is the linear combination of the responses of the circuits to the individual stimuli.

In this section will be described a class of linear circuits, known as autonomous linear feedback shift registers that have the canonical form shown in Figures 9.1 and 9.2. Here $c_i$ is a binary constant and $c_i$ = 1 implies that a connection exists, while $c_i$ = 0 implies that no connection exists. When $c_i$ = 0 the corresponding XOR gate can be replaced by a direct connection from its input to its output.

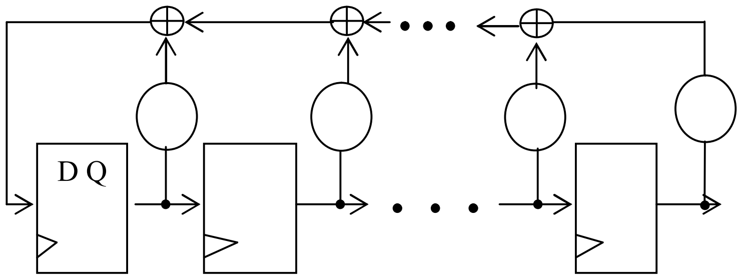


Figure 9.1. Type 1 (external- XOR) LFSR

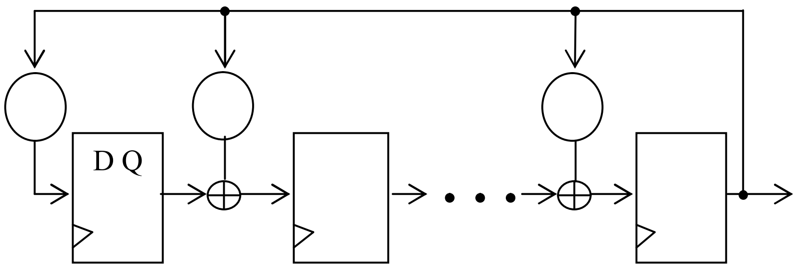


Figure 9.2. Type 2 (internal-XOR) LFSR

## 3.2 CHARACTERISTIC POLYNOMIALS

A sequence number $a_0, a_1, \ldots, a_m, \ldots$ can be associated with a polynomial, called a generating function *G(x)*, by the rule

$$G(x)= a_0 + a_1 x + a_2 x^2 + .. +a_m x^m \ldots .$$

Let $\{ a_m \}= a_0, a_1, \ldots$ represent the output sequence generated by an LFSR, where $a_i = 0$ or 1. Then this sequence can be expressed as

$$G(x) = \sum_{m=0}^{\infty} a_m x^m \qquad (1)$$

For the type 1 LFSR, it could be shown, that if the current state of $Q_i$ is $a_{m-1}$, for $i$=1, 2, …, $n$, then

$$a_m = \sum_{i=1}^{n} c_i a_{m-i} \qquad (2)$$

Thus recurrence relation can define the operation of the circuit. Let the initial state of LFSR be $a_{-1}, a_{-2}, \ldots, a_{-n+1}, a_{-n}$. The operation of the circuit starts $n$ clock periods before generating the output $a_0$. Since

$$G(x) = \sum_{m=0}^{\infty} a_m x^m$$

Substituting for $a_m$ we get

$$G(x)=\sum_{m=0}^{\infty}\sum_{i=1}^{n}c_i a_{m-i}x^m=\sum_{i=1}^{n}c_i x^i\sum_{m=0}^{\infty}a_{m-i}x^{m-i}=$$
$$=\sum_{i=1}^{n}c_i x^i\left[a_{-i}x^{-i}+\ldots+a_{-1}x^{-1}+\sum_{m=0}^{\infty}a_m x^m\right]=\sum_{i=1}^{n}c_i x^i\left[a_{-i}x^{-i}+\ldots+a_{-1}x^{-1}+G(x)\right].$$

Hence

$$G(x)=\sum_{i=1}^{n}c_i x^i G(x)+\sum_{i=1}^{n}c_i x^i\left(a_{-i}x^{-i}+\ldots+a_{-1}x^{-1}\right)$$

Or

$$G(x)=\frac{\sum_{i=1}^{n}c_i x^i\left(a_{-i}x^{-i}+\ldots+a_{-1}x^{-1}\right)}{1+\sum_{i=1}^{n}c_i x^i} \qquad (3)$$

Thus *G (x)* is a function of the initial state $a_{-1}, a_{-2}, \ldots, a_{-n+1}, a_{-n}$ of the LFSR and the feedback coefficients $c_1, c_2, \ldots, c_n$. The denominator in (3), denoted by

$$P(x)=1+c_1x+c_2x^2+\ldots+c_nx^n$$

Is referred to as the characteristic polynomial of the sequence $[a_m]$ and of the LFSR. For an *n*-stage LFSR, $c_n$=1. Note that *P (x)* is only a function of the feedback coefficients. If we set $a_{-1}= a_{-2}= \ldots = a_{1-n}=0$, and $a_{-n}=1$, then (3) reduces to

$$G(x)=\frac{1}{P(x)}$$

Thus the characteristic polynomial along with the initial state characterizes the cyclic nature of an LFSR and hence characterizes the output sequence. So *for* $a_{-1}= a_{-2}= \ldots = a_{1-n}=0$, and $a_{-n}=1$, then function *G (x)* is follow:

$$G(x)=\frac{1}{P(x)}=\sum_{m=0}^{\infty}a_m x^m \qquad (4)$$

## 3.3 PERIODICITY OF LFSRs

As it was told before an LFSR goes through a cyclic or periodic sequence of states and that the output produced is also periodic. The maximum length of this period is $2^{n-1}$, where *n* is the number of stages. In this section we consider properties related to the period of an LFSR. Most results will be presented without proof.

If the initial state of an LFSR is $a_{-1} = a_{-2} = \ldots = a_{1-n} = 0$, and $a_{-n}=1$, then the LFSR sequence $[a_m]$ is periodic with a period that is the smallest integer $k$ for which $P(x)$ divides $(1-x^k)$.

*Maximum-length sequence* is the sequence generated by an $n$-stage LFSR has period $(2^n-1)$
*Primitive polynomial* is the characteristic polynomial associated with a maximum-length sequence.

*An irreducible polynomial* is one that cannot be factored, because it is not divisible by any other polynomial other than 1 and itself.

*An irreducible polynomial P (x)* of degree $n$ satisfies the following two conditions:

1. For $n \geq 2$, $P(x)$ has an odd number of terms including the 1 term.
2. For $n \geq 4$, $P(x)$ must divide (evenly) into $(1+x^k)$, where $k=(2^n-1)$.

*An irreducible polynomial* is primitive if the smallest positive integer $k$ that allows the polynomial to divide evenly into $(1+x^k)$ occurs for $k=(2^n-1)$, where $n$ is the degree of the polynomial.

The number of primitive polynomials for $n$-stage LFSR is given by the next formula

$$\lambda_2(n)=\Phi(2^n-1)/n$$

where

$$\Phi(n) = n\prod_{p|n}\left(1-\frac{1}{p}\right)$$

and $p$ is taken over all primes that divide $n$. Table 3 shows some values of $\lambda_2(n)$.

| N | $\lambda_2(n)$ |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 4 | 2 |
| 8 | 16 |
| 16 | 2048 |
| 32 | 67108864 |

Table 3. Number of primitive polynomials of degree n

## 3.4 CHARACTERISTICS OF MAXIMUM–LENGTH SEQUENCES

Sequences generated by LFSRs that are associated with a primitive polynomial are called pseudorandom sequences, since they have many properties like those of random sequences. However, since they are periodic and deterministic, they are pseudorandom, not random. Some of these properties are listed next.

In the following, any string of *$(2^n - 1)$* consecutive outputs is referred to as an m-sequence.

*Property 1*. The number of 1s in an *m*-sequence differs from the number of 0s by one.
*Property 2*. An *m*-sequence produces an equal number of runs of 1s and 0s.
*Property 3*. In every *m*-sequence, one half the runs have length 1, one fourth have length 2, one eighth have length 3, and so forth, as long as the fractions result in integral numbers of runs.

These properties of randomness make feasible the use of LFSRs as test sequence generators in BIST circuitry.

## 3.5 LFSRs USED AS SIGNATURE ANALYZERS

Signature analysis is a compression technique based on the concept of cyclic redundancy checking (CRC). In the simplest form of the scheme shown below, the signature generator consists of a single-input LFSR. The signature is the contents of this register after the last input bit has been sampled. Figure 10 illustrates this concept.

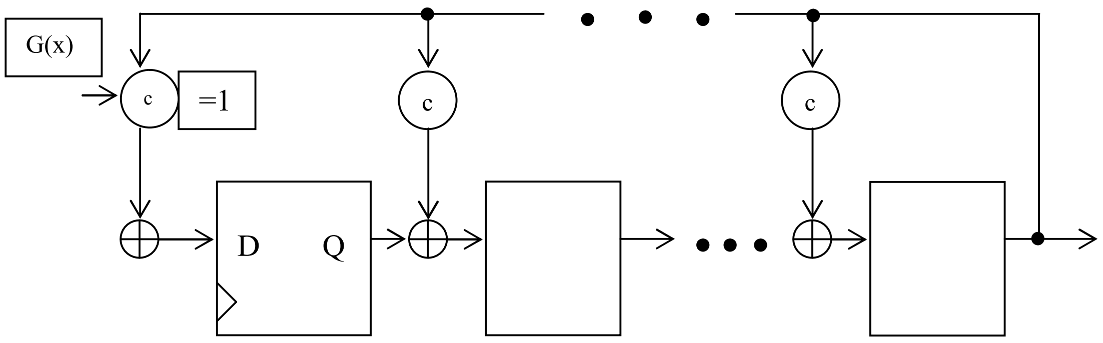


Figure 10. A type 2 LFSR used as a signature analyzer

It is possible, that we get a signature of faulty circuit same as the normally functioning one, this effect called errors masking. The proportion of error streams that mask to the correct signature *$S(R_0)$* is independent of actual signature. For a test bit stream of length *m*, there are *$2^m$* possible response streams, one of which is correct. The number of bit streams that produce a specific signature is

$$\frac{2^m}{2^n} = 2^{m-n}$$

where the LFSR consists of *n* stages and the all-zero state is now possible because of the existence of an external input. For a particular fault-free response, there are *$(2^{m-n} - 1)$* erroneous bit streams that will produce the same signature. Since there are a total of *$(2^m - 1)$* possible erroneous response streams, the proportion of masking error stream is

$$P_{SA}(M|m,n) = \frac{2^{m-n}-1}{2^m-1} \approx 2^{-n}$$

where, the approximation holds for $m>>n$

If all possible error streams are equally likely, which is rarely the case, then $P_{SA}(M|m,n)$ is probability that an incorrect response will go undetected, i.e., the probability of no masking $(1-2^{-n})$. This is somewhat strange result since it is only a function of the length of the LFSR and not of the feedback network. Increasing the register length by one stage reduces the masking probability by a factor of 2. Note that because of the feedback network, all single- bit errors are detectable. However, there is no direct correlation between faults and error masking. Thus a 16-bit signature analyzer may detect $100(1-2^{-16})$= 99.9984 percent of the erroneous responses but not necessarily this same percentage of faults.

Signature analysis is the most popular method employed for test data compression because it usually produces the smallest degree of masking.

## 3.6 SHIFT REGISTER POLYNOMIAL DIVISION

The theory behind the use of an LFSR for signature analysis is based on the concept of polynomial division, where the "remainder" left in the register after completion of the test process corresponds to the final signature.

Consider to the type 2(internal-XOR) LFSR shown in Figure 10. The input sequence $[a_m]$ can be represented by the polynomial *G (x)* and the output sequence by *Q (x)* The highest degree of the polynomials *G (x)* and *Q (x)* corresponds, respectively, to the first input bit to enter LFSR and the first output bit produced n clock periods later, where *n* is the degree of the LFSR. If the initial state of the LFSR is all zeros, let the final state of the LFSR be represented by the polynomial *R (x)*. Then it can be shown that these polynomials are related by the equation

$$\frac{G(x)}{P^*(x)} = Q(x) + \frac{R(x)}{P^*(x)}$$

where *P*(x)* is the reciprocal characteristic polynomial of the LFSR. The reciprocal characteristic polynomial is used because $a_m$ corresponds to the first bit of the input stream rather than the last bit. Type 1 (external-XOR) LFSRs also carry out polynomial division and produce the correct quotient. However, the contents of the LFSR are not the remainder as is for the type 2 LFSRs. But it can be shown that all input sequences, which are equal to each other modulo *P (x)*, produce the same remainder.

## 3.7 ERROR POLYNOMIAL AND MASKING

Let *E(x)* be an error polynomial, where each non-zero coefficient represents an error occurring in the corresponding bit position. As an example, let the correct response be $R_0$ =10111 and the erroneous response be $R'(x)$=11101. Then the difference or error polynomial is 01010. Thus $G_0(x) = x^4 + x^2 + x + 1$, $G'(x) = x^4 + x^3 + x^2 + 1$, and

*E(x)=$x^3$ + x*. Clearly *G' (x)= G (x) +E (x)*(modulo 2). Since *G (x)=Q (x) P* (x) + R (x)*, an undetectable response sequence is one that satisfies the equation

$$G'(x) = G(x) + E(x) = Q'(x)P^*(x) + R(x)$$

*G' (x)* and *G (x)* produce the same remainder. From this observation we obtain the following well-known result from algebraic coding theory.

Let *R (x)* be the signature generated for an input *G (x)* using the characteristic polynomial *P (x)* as a divisor in a LFSR. For an error polynomial *E (x)*, *G (x) G' (x)=G (x)+E (x)* have the same signature *R (x)* if and only if *E (x)* is a multiple of *P (x).*

Thus both type 1 and type 2 LFSRs can be used to generate the signature *R (x)*. Henceforth, the final contents of the LFSR will be referred to as the signature, because sometimes, depending on the initial state and polynomial *P (x),* the final state does not correspond to the remainder of *G (x)/P (x).*

For an input data stream of length *m*, if all possible error patterns are equally likely, then the probability that an *n*-bit signature generator will not detect an error is

$$P(M) = \frac{2^{m-n} - 1}{2^m - 1}$$

which, for *m>>n*, approaches $2^{-n}$.

This result follows directly from the previous theorem because *P (x)* has *(*$2^{m-n} - 1$*)* non-zero multiples of degree less than *m*. It also corresponds to the same result given earlier but based on a different argument. Note that this result is independent of the polynomial *P (x)*. This includes *P (x)=*$x^n$ , which has no feedback, it is just a shift register. For this case, the signature is just the last *n* bits of the data stream. In fact one can use the first *n* bits and truncate the rest of test sequence and obtain the same results. These strange conclusions follow from the assumption that all error patterns are equally likely. But in this case long test sequences would not be necessary.

To see why this assumption is done, consider a minimal-length test sequence of length *m* for a combinational circuit. Clearly the *i*-th test vector $t_i$ detects some fault $f_i$ not detected by $t_j$*, j =1, 2, ..., i-1.* Thus if $f_i$ is present in the circuit, the error pattern is of the form 00 … 01xx …, the first *i-1* bits must be *0.* Several other arguments can be made to show that all error patterns are not equally likely.

An LFSR signature analyzer based on any polynomial with two or more non-zero coefficients detects all single-bit errors. Assume *P (x)* has two or more non-zero coefficients. Then all non-zero multiples of *P(x)* must have at least two non-zero coefficients. Hence an error pattern with only one non-zero coefficient cannot be multiple of *P (x)* and must be detectable.

A *(k, k) burst error* is one where all erroneous bits are within *k* consecutive bit position, and at most *k* bits are in error. If *P (x)* is of degree *n* and the coefficients of $x^0$ is *1*, then all *(k, k)* burst errors are detected as long as $n \geq k$. Rather than assuming that all error patterns are equally likely, one can assume that the probability that a response bit is in error is *p*. Then for *p=0,5* the probability of masking is $2^{-n}$. For very small or very large values of *p*, the probability of masking approaches the value

$$2^{-n} + (2^n - 1)(1 - 2p)^{m(1-1/(2^n-1))}$$

where *m* is the length of the test sequence.
Experimental results also show that using primitive polynomials helps in reducing masking effect.

In conclusion, the bound of $2^{-n}$ on error masking is not too useful since it is based on realistic assumptions. However, in general, signature analysis gives excellent results. Results are sensitive to *P (x)* and improve as *n* increases. Open problems deal with selecting the best characteristic polynomial *P (x)* to use, characterizing error patterns, correlating fault coverage to *P (x)* and determining the probability of masking.

## 3.8 MULTIPLE-INPUT SIGNATURE REGISTER

Signature analysis can be extended to testing multiple-output circuits. Normally a single-output signature analyzer is not attached to every output because of the resulting high overhead. A single signature analyzer could be time-multiplexed, but that would require repeating the test sequence for each output, resulting in a potentially long test time. The most common technique is to use a multiple-input signature register (MISR), such as the one shown in Figure 11. Here we assume that the CUT has *n* (or less) outputs. It is seen that this circuit operates as *n* single-input signature analyzer. For example, by setting $D_i = 0$ for all $i \neq j$, the circuit computes the signature of the data entering on line $D_j$. The mathematical theory associated with MISRs will not be presented, but it follows as a direct extension of the results presented previously for Single Input Signature Registers (SISR). An error pattern can be associated with each input $D_i$. These error patterns are merged within LFSR. Again, assuming all error patterns are equally likely, the probability that a MISR will not detect an error is approximately $2^{-n}$.

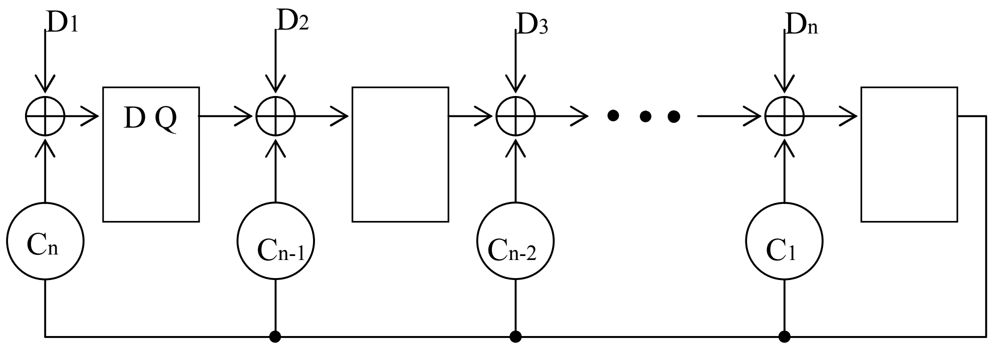


Figure 11. Multiple-input signature register

## 3.9 SELECTION OF THE POLYNOMIAL P (X)

As stated previously, a MISR having *n* stages has a masking probability approximately equal to $2^{-n}$ for equally likely error patterns long data streams. Also, this result is independent of *P (x)*. Let the error bit associated with $D_i$ at time *j* be denoted by $e_{ij}$, where *i*= 1, 2, ... , n, and *j* = 1, 2, …, *m*. Then the error polynomial associated with $D_i$ is

$$E_i = \sum_{j=1}^{m} e_{ij} x^{j-1}$$

Then the effective error polynomial is

$$E(x) = \sum_{i=1}^{n} E_i x^{i-1}$$

assuming that the initial state of the register is all zeros. The error polynomial *E (x)* is masked if it is a multiple of *P (x)*. So a complex-feedback LFSR structure is typically used on the assumption that it will reduce the chances of masking an error.

When the characteristic polynomial is the product of the parity generator polynomial *g(x)= x+1* and a primitive polynomial of degree *(n-1)*, an *n*-stage MISR has the property that the parity over all the bits in the input streams equals the parity of the final signature. Hence masking will not occur for an odd number of errors.

## 3.10 INCREASING THE EFFECTIVENESS OF SIGNATURE ANALYSIS

There are several ways to decrease the probability of masking. Based on the theory presented, the probability of masking can be reduced be increasing the length of the LFSR. Also a test can be repeated using a different feedback polynomial. When testing combinational circuits, a test can be repeated after first changing the order of the test vectors, thus producing a different error polynomial. This technique can be also used for sequential circuits, but now the fault-free signature also changes.

Masking occurs because once an error exists within an LFSR, it can be canceled by new errors occurring on the inputs. Inspecting the contents of the signature analyzer several times during the testing process decreases the chance that a faulty circuit will go undetected. This technique is equivalent to periodically sampling the output of the signature analyzer. The degree of storage compression is a function of how often the output is sampled. [1]

## 3.11 CONCLUDING LFSRs THEORY

Using linear feedback registers for test pattern generating and as a response compactors is widely used since they are easy to implement, they can be used for field test and self-testing, and can provide high fault coverage, though the correlation between error coverage and fault coverage is hard to predict.

Signature analysis is widely used because it provides excellent fault and error coverage, though fault coverage must be determined using a fault simulator or a statical fault simulator. Unlike the other techniques, several means exist for improving the coverage without changing the test, such as by changing the characteristic polynomial or increasing the test length of the register.

# 4 BUILT-IN SELF-TEST (BIST)

## 4.1 INTRODUCTION TO BIST CONCEPTS

The main idea of Built-in self-test (BIST) is the capability of a circuit (chip, board, or system) to test itself.

BIST techniques can be classified into two categories, namely on-line BIST, which includes concurrent and no concurrent techniques, and off-line BIST, which include functional and structural approaches (Figure 12).

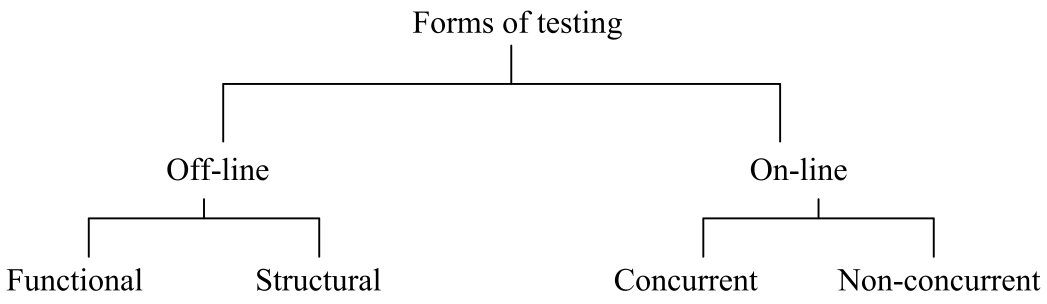


Figure 12. Forms of testing

In on-line BIST, testing occurs during normal functional operating conditions; i.e., the circuit under test (CUT) is not placed into a test mode where normal functional operation is locked out. Concurrent on-line BIST is a form of testing that occurs simultaneously with normal functional operation. In no concurrent on-line BIST, testing is carried out while a system is in an idle state. This is often accomplished by executing diagnostic software routines or diagnostic firmware routines. The test process can be interrupted at any time so that normal operation can resume.

Off-line BIST deals with testing a system when it is not carrying out its normal functions. Systems, boards, and chips can be tested in this mode. This form of testing is also applicable at the manufacturing, field and operational levels. Often Off-line testing is carried out using on-chip or on-board test-pattern generators (TPGs) and output response analyzers (ORAs) or microdiagnostic routines. Off-line testing does not detect errors in real time, i.e., when they first occur, as is possible with many on-line concurrent BIST techniques.

Functional off-line BIST deals with the execution of a test based on a functional description of the CUT and often employs a functional, or high-level, fault model. Normally such a test is implemented as diagnostic software or firmware.

Structural off-line BIST deals with the execution of a test based on the structure of the CUT. An explicit structural fault model may be used. Fault coverage is based on detecting structural faults. Usually tests are generated and responses are compressed using some form of an LFSR. The theory of LFSR will be described later. In the next section various forms of testing and related TPGs will be described.

## 4.2 TEST-PATTERN GENERATION FOR BIST

### 4.2.1 EXHAUSTIVE TESTING

Exhaustive testing deals with the testing of an n-input combinational circuit where all $2^n$ inputs are applied. A binary counter can be used as TPG. If a maximum-length autonomous LFSR is used, its design can be modified to include the all-zero state.

Exhaustive testing guarantees that all the detectable faults that do not produce sequential behavior will be detected. Depending on the clock rate, this approach is usually not feasible if $n$ is larger than about *22*. Other techniques to be described are more practical when $n$ is large. The concept of exhaustive testing is not generally applicable to sequential circuits.

### 4.2.2 PSEUDORANDOM TESTING

Pseudorandom testing deals with testing a circuit with test patterns that has many characteristics of random patterns but where the patterns are generated deterministically and hence are repeatable. Pseudorandom patterns can be generated with or without replacement. Generation with replacement implies that each pattern is unique. Not all $2^n$ test patterns need be generated. Pseudorandom test patterns without replacement can be generated by an autonomous LFSR. Pseudorandom testing is applicable to both combinational and sequentional circuits. Fault coverage can be determined by fault simulation. The test length is selected to achieve an acceptable level of fault coverage. Unfortunately, some circuits contain random-pattern-resistant faults and thus require long test length to insure high fault coverage.

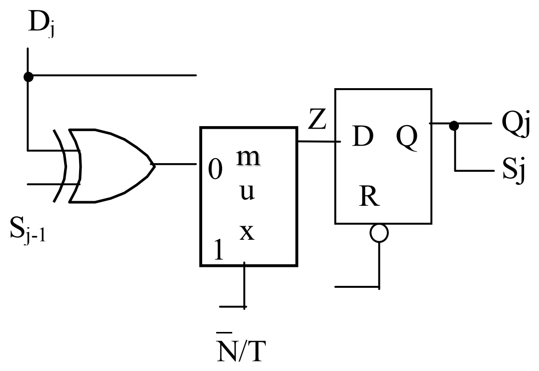
### 4.2.3 PSEUDOEXHAUSTIVE TESTING

Pseudoexhaustive testing achieves many of the benefits of exhaustive testing but usually requires far fewer test patterns. It relies on various forms of circuit segmentation and attempts to test each segment exhaustively.

There are several forms of segmentation, a few of which are listed below:
1. Logical segmentation
    - Cone segmentation
    - Sensitized path segmentation
2. Physical segmentation

## 4.3 CIRCULAR SELF-TEST PATH (CSTP)

The circular self –test path (CSTP) is intended for register- based BIST architecture, where self-test cells are grouped into registers, CSTP employs a self-test design shown in Figure 13 and partial self-test, where not all registers must consist of self-test cells. Some necessary features of this architecture are all inputs and outputs must be associated with boundary scan cells and all storage cells must be initializable to a known state before testing.



| $\overline{N}/T$ | Z | Mode |
|---|---|---|
| 0<br>1 | $D_j$<br>$D_j \oplus S_{j-1}$ | System<br>Test |

Figure 13. Storage cell design for use in CSTP BIST architectures.

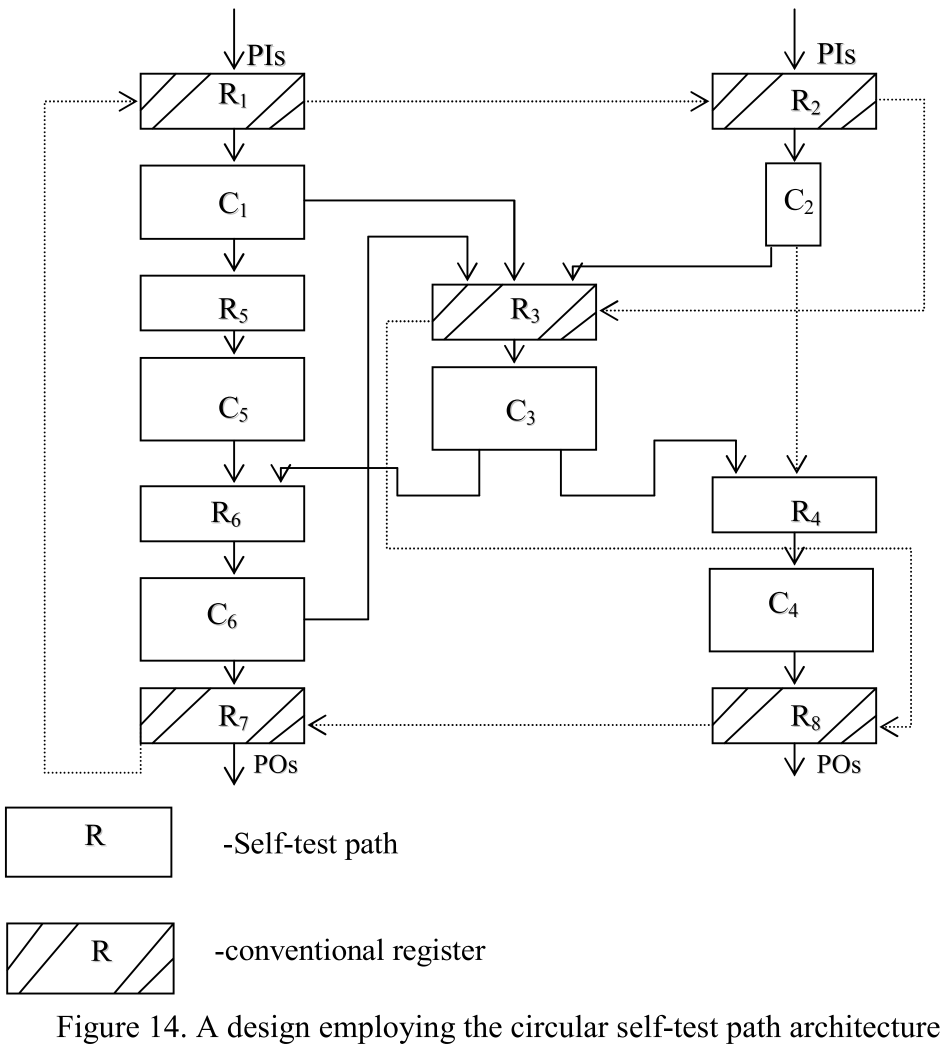


Figure 14. A design employing the circular self-test path architecture

Consider the circuit shown in Figure 14. The registers $R_1$, $R_2$, $R_3$, $R_7$ and $R_8$ are part of the circular self-test path. Note that the self-test cells form a circular path. If this circular path contains *m* cells, then it corresponds to a MISR having the characteristic polynomial $(1+x^m)$. Registers $R_4$, $R_5$ and $R_6$ need not to be in the self-test path, nor do they require reset or set lines, since they can be initialized based on the state of the rest of the circuit. That is, once $R_1$, $R_2$ and $R_3$ are initialized, if $R_4$, $R_5$, and $R_6$ are issued two system clocks, then they too will be initialized to known state

The same cannot be said of $R_3$ because of the feedback loop formed by the path $R_3$-$C_3$-$R_6$-$C_6$-$R_3$. However, if $R_3$ has a reset line, then it needs not to be in the self-test path. Increasing the number of cells in the self-test path increases both the BIST hardware overhead and the fault coverage for a fixed test length.

The test process requires three phases.

1. *Initialization:* All registers are placed into a known state
2. *Testing of CUT*: The circuit is run in the test mode; registers that are not in the self-test path operate in their normal mode.
3. *Response evaluation.*

During phase 2 the self-test path operates as both a random-pattern generator and response compactor. During phase 3 the circuit is again run in the test mode. But now the sequences of outputs from one or more self-test cells are compared with precomputed fault-free values. This comparison can be done either on-chip or off-chip.

Figure 15 shows the general form of a circular self-test path design. The circular path corresponds to an 'LFSR having the primitive polynomial p $(x)=1+x^m$. In this section some theoretical and experimental results about certain performance aspects of this class of design will be briefly presented. These results are applicable to many designs where a MISR is used as PRPG.

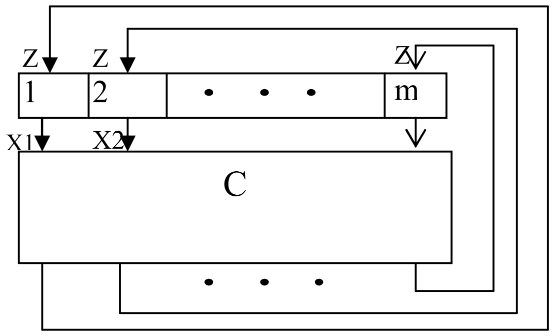


Figure 15. General form of a circular self-test path design

Let $z_i\ (t)$ and $x_i\ (t)$ be the input and output, respectively, to the *i*th cell in circular self-test path. Then assume that the sequences of bits applied to each cell in the path are independent and that each sequence is characterized by a constant (in time) probability of a 1, for input $z_i\ (t)$, $p_i = Prob\ \{z_i\ (t)=1\}$, $t = 1, 2, \ldots$.

If there exists an input to the circular path, $z_i$, such that $0 < p_i < 1$, then independent of the initial state of the path, $\lim_{t\to\infty} prob\{x_j(t)=1\} = 0.5$, $j = 1, 2, \ldots, m$.

Thus if response of the circuit to the initial state of the circular path is neither the all-zeros nor the all-ones pattern, then some time after initialization the probability of a 1 at any bit position of the circular path is close to 0,5.

The number of clock cycles required for $x_i(t)$ to converge to 0,5 is a function of the length of the circular path, and is usually small compared to the number of test patterns normally applied to the circuit.

The pattern coverage (also known as the state coverage) is denoted by $C_{n,r}$ and is defined as the fraction of all $2^n$ binary patterns occurring during *r* clock cycles of the self-testing process at *n* arbitrary selected outputs of the circular path. These outputs can be the *n* inputs to a block of logic C.

As the length of the circular path increases, the impact of the value of $p_i$ on the pattern coverage decreases.

The circular path provides a block C with an almost exhaustive test for test lengths a few times longer than an exhaustive test.

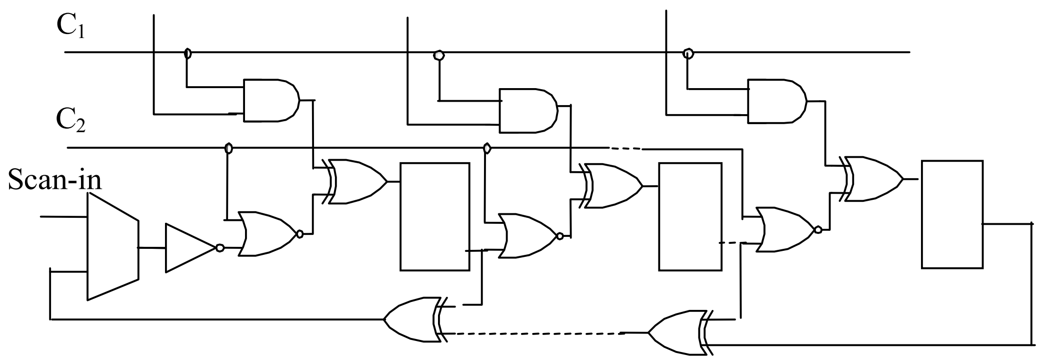
When the number of clock cycles associated with a test exceeds the length of the circular path, the impact of the location of the n cells feeding the block C on the pattern coverage is negligible. For long test times, the pattern coverage associated with *n* cells is almost independent of the length of the path.

## 4.4 BUILT-IN LOGIC BLOCK OBSERVATION (BILBO)

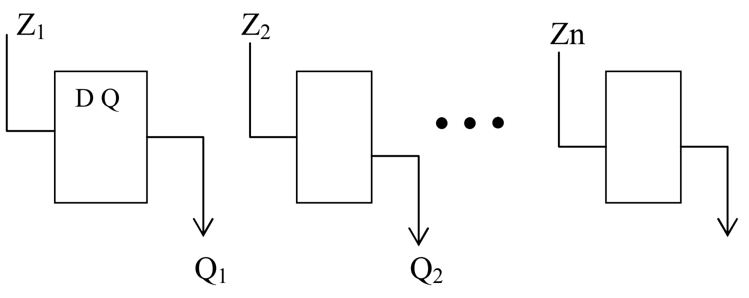
One major problem of the most BIST designs is that they deal with an unpartioned version of a CUT, where all primary inputs are grouped together into one set, all primary outputs into a second set, and all storage cells into a third set. These sets are then associated with PRPGs and MISRs. Since the number of cells in these registers is usually large, it is not feasible to consider exhaustive or pseudoexhaustive test techniques. For example, a chip can easily have over 100 inputs and several hundred storage cells. To circumvent this problem one can attempt to cluster storage cells into groups, commonly called registers. In general these groups correspond to the functional registers found in many designs, such as the program counter and the instruction register. Some BIST architectures take advantage of the register aspects of many designs to achieve a more effective test methodology.

One such architecture employs built-in logic-block observation (BILBO) registers, shown in Figure 16(a). In this register design the inverted output $\overline{Q}$ of a storage cell is connected via a NOR and a XOR gate by the data input of the next cell. A BILBO register operates in one of four modes, as specified by the control inputs $B_1$ and $B_2$. When $B_1 = B_2 = 1$, the BILBO register operates in its normal parallel load mode (see Figure 16 (b)). When $B_1 = B_2 = 0$, it operates as a shift register with scan input $S_i$ (see Figure 16(c)). Note that the data is complemented as it enters the scan register. When $B_1=0$ and $B_2=1$, all storage cells are reset. When $B_1=1$ and $B_2=0$, the BILBO register is configured as an LFSR (see Figure 16(d)), or more accurately the register operates as a MISR. If the $Z_i$s are the outputs of a CUT, then the register compresses the response to form a signature. If the inputs $Z_1, Z_2, \ldots, Z_n$ are held at a constant value of *0*, and the initial value of the register is not all-zeros, then the LFSR operates as a pseudorandom-pattern generator.

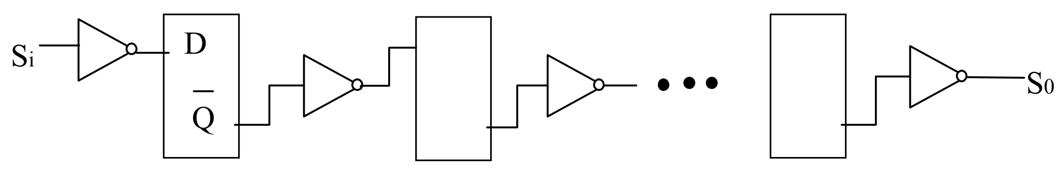
A simple form of a BILBO BIST architecture consists of partitioning a circuit into a set of registers and blocks of combinational logic, where the normal registers are replaced by BILBO registers. In addition, the inputs to a block of logic C are driven by a BILBO register $R_i$, and the outputs of C drive another BILBO register $R_j$.



(a)



(b)



(c)

Qn

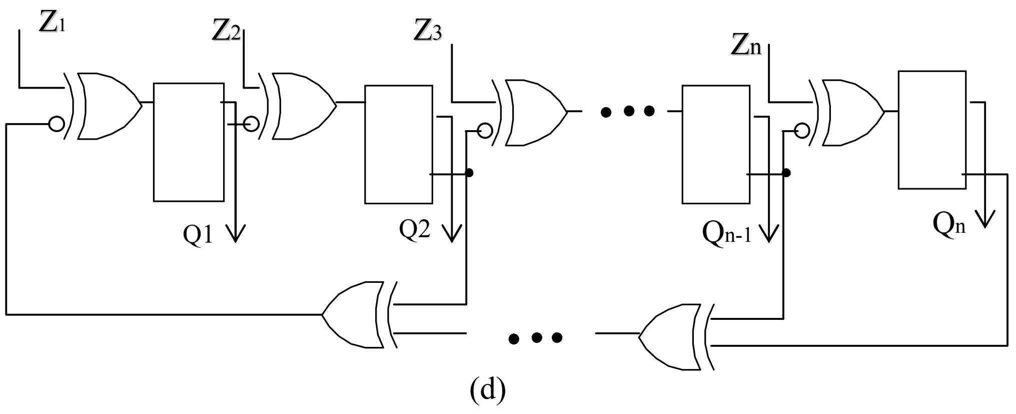


Figure 16. n-bit BILBO register

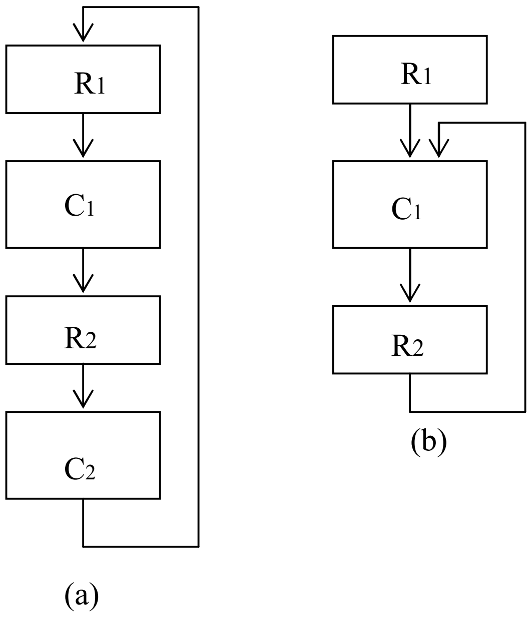


Figure 17. BIST designs with BILBO registers

Consider the circuit shown in Figure 17(a), where the registers are all BILBOs. To test $C_1$, first $R_1$ and $R_2$ are seeded, and then $R_1$ is put into the PRPG mode and $R_2$ into the MISR mode. Assume the inputs of $R_1$ are held at the value 0. The circuit is then run in this mode for N clock cycles. If the number of inputs of $C_1$ is not too large, $C_1$ can even be tested exhaustively, except for the all-zero pattern. At the end of this test process, called a test session, the contents of $R_2$ can be scanned out and the signature checked. Similarly configuring R1 to be a MISR and R2 to be a PRPG can test C2. Thus the circuit is tested in two test sessions.

Figure 17(b) shows a different type of circuit configuration, one having a self-loop around the $R_2$ BILBO register. This design does not conform to a normal BILBO architecture. To test $C_1$, $R_1$ must be in the PRPG mode. This is not possible for the design shown in Figure 16(a). What can be done is to place $R_2$ in the MISR mode. Now its outputs are essentially random vectors that can be used as test data to $C_1$. One feature of this scheme is that errors in the MISR produce "erroneous" test patterns that are applied to $C_1$, which is tend to produce more errors in $R_2$. The bad aspect of this approach is that there may exist faults that are never detected. This could occur, for example, if the input data to $C_1$ never propagate the effect of a fault to the output of $C_1$.

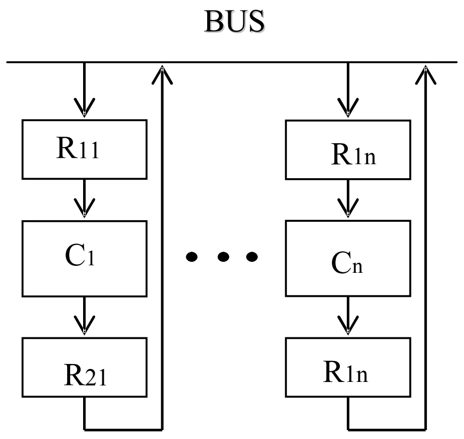


Figure 18. Bus-oriented BIST architecture

The situation can be rectified by using a concurrent built-in logic-block observation (CBILBO) register. This register operates simultaneously as a MISR and a PRPG. Recall that when a BILBO register is in the PRPG mode, its inputs need to be held at some constant value. This can be achieved in several ways. Often the BILBO test methodology is applied to a modular and bus-oriented system in which functional modules, such as ALUs, RAMs, ROMs, are connected via a register to a bus (see Figure 18). By disabling all bus drivers and using pull-up or pull-down circuitry, the register inputs can be held in a constant state.

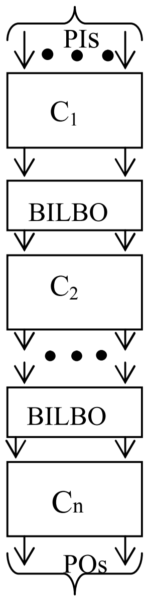


Figure 19. Pipeline-oriented BILBO architecture

However, some architecture has a pipeline structure shown as in Figure 19. To deactivate the inputs to a BILBO register during its PRPG mode, a modified BILBO register design having three control states, and one can be used to specify the MISR mode and another the PRPG mode.

One aspect that differentiates the BILBO architecture from the other BIST architectures is the partitioning of storage cells to form registers and the partitioning of the combinational logic into blocks of logic. Other types of registers, such as constant-weight counters or more complex forms of LFSRs can replace the BILBO registers.[1]

# 5 FUNCTIONAL TESTING

In this section will be described functional testing methods that are based on functional model of the system.

## 5.1 INTRODUCTION TO FUNCTIONAL TESTING

A functional model reflects the functional specifications of the system and, to a great extent, is independent of its implementation. Therefore functional tests derived from a functional model can be used only to check whether physical faults are present in the manufactured system, but also as design verification tests for checking that the implementation is free of design errors.

The objective of functional testing is to validate the correct operation of a system with respect to its functional specifications. This can be approached in two different ways. One approach assumes specific functional fault models and tries to generate tests that detect the faults defined by these models. By contrast, the other approach is not concerned with the possible types of faulty behavior and tries to derive tests based only on the specified fault-free behavior. Between these two there is a third approach that defines an implicit fault model, which assumes that almost any fault can occur. Functional tests detecting almost any fault are said to be exhaustive, as they must completely exercise the fault-free behavior. Because of the length of the resulting tests, exhaustive testing can be applied in practice only to small circuits. By using some knowledge about the structure of the circuit and by slightly narrowing the universe of faults are guaranteed to be detected, we can obtain pseudoexhaustive tests that can be significantly shorter than the exhaustive ones.

## 5.2 EXHAUSTIVE AND PSEUDOEXHAUSTIVE TESTING

*The Universal Fault Model*

Exhaustive tests detect all the faults defined by the universal fault model. This implicit fault model assumes that any fault is possible, except those that increase the number of states in a circuit. For a combinational circuit N realizing the function Z(x), the universal fault model accounts for any fault f that changes the function to $Z_f(x)$.The only faults not included in this model are those that transform *N* into a sequential circuit. For a sequential circuit, the universal fault model accounts for any fault that changes the state table without creating new states.

To test all the faults defined by the universal fault model in a combinational circuit with n primary inputs, we need to apply $2^n$ possible input vectors. The exponential growth of the required number of vectors limits the practical applicability of this exhaustive testing method only to circuit with less than *20* primary inputs. Further pseudoexhausive testing methods will be presented.

### 5.2.1 PARTIAL-DEPENDENCE CIRCUITS

Let $O_1, O_2, \dots, O_m$ be the primary outputs of a circuit with *n* primary inputs, and let $n_i$ be the number of primary inputs feeding $O_i$. A circuit in which no primary outputs depend on all the primary inputs, is said to be a partial-dependence circuit. For such circuit, pseudoexhaustive testing consists in applying all $2^{n_i}$ combinations to the $n_i$ inputs feeding every primary output $O_i$. .

## 5.2.2 PARTITIONING TECHNIQUES

The pseudoexhaustive testing techniques described in the previous section are not applicable to total-dependence circuits, in which at least one primary output depends on all primary inputs. Even for a partial-dependence circuit, the size of a pseudoexhaustive test set may still be too large to be acceptable in practice. In such cases, pseudoexhaustive testing can be achieved by partitioning techniques. The principle of is to partition the circuit into segments such that the number of inputs of every segment is significantly smaller than the number of primary inputs of the circuit. Then the segments are exhaustively tested. The main problem with this technique is that, in general, the inputs of a segment are not primary inputs and its outputs are not primary outputs.
Then we need a means to control the segment inputs from the primary inputs and to observe its outputs at the primary outputs. One way to achieve this, referred to as sensitized partitioning, is based on sensitizing path from primary inputs to the segment inputs and from the segment outputs to primary outputs.

## 5.3 FUNCTIONAL BIST

Functional BIST is a promising solution for self-testing complex digital systems at reduced costs in terms of area and performance degradation.

To increase the functionality, achieve higher performance, and decrease cost, designers are actually moving quickly towards very deep sub-micron technologies. From one hand, the vast availability of gates permits the integration of a variety of memories, processors and analog units on a single chip. On the other hand traditional testing approaches based on an external ATE become more and more unfeasible. The number of externally accessible I/O pins counting up to several hundreds strongly limits the controllability and observability of embedded cores.

In traditional BIST architectures, LFSRs and multifunctional registers, like BILBO, mostly perform test pattern generation.

The functional BIST strategy, exploits functionality's and modules embedded into the system itself for test pattern generation. In Figure 20 both modules $M_i$ and $M_j$ are part of the system logic. And during testing $M_i$ is controlled in such way that outputs serve as a test patterns for module $M_j$. Typically $M_i$ is a sequential circuit used as test pattern generator (STPG) for a given unit under test (UUT), in this case $M_j$.

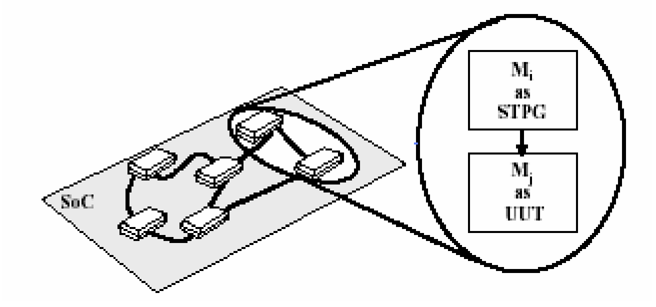


Figure 20. Functional BIST

Further will be described the universal method to control and initialize sequential structures so that they work as an STPG for a given unit under test.

The sequence of values appearing on the STPG is a function of the triplet (σ, δ, τ) as well as of the logic function embedded into the block. This is shown in Figure 21. First the state register of the STPG is initialized with initial state (δ) and its primary inputs are fixed at an input value (σ), then the STPG is let evolve for a certain number of clock cycles (τ). Initial state and the input value are often collectively referred to as seed of the STPG.

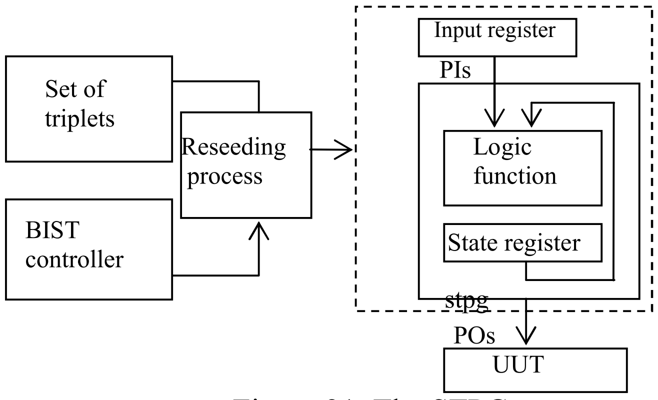


Figure 21. The STPG

For sake of efficiency and flexibility, the STPG can be periodically reseeded, stopping its evaluation and restarting it with a new triple (σ, δ, τ) until the target fault coverage is reached. In such a case, the global test length is a function of the number of reseeding:

$$\sum_{0\le i\le n} \tau_i$$

When adopting functional approach, the designer can trade-off between:
- Maximizing the fault coverage.
- Minimizing the test times, global test length

Minimizing the complexity of the BIST controller, by selecting a proper common τ.

# 6 APPLET "DESIGN AND TEST OF DIGITAL SYSTEMS ON RT-LEVEL" DESCRIPTION

The applet is introduced as the teaching system wich shows how to provide digital design on RT-level and also shows a variety of different modern testing techniques including functional and deteministic testing, a number of BIST solutions.

**Overall description of the teaching system**

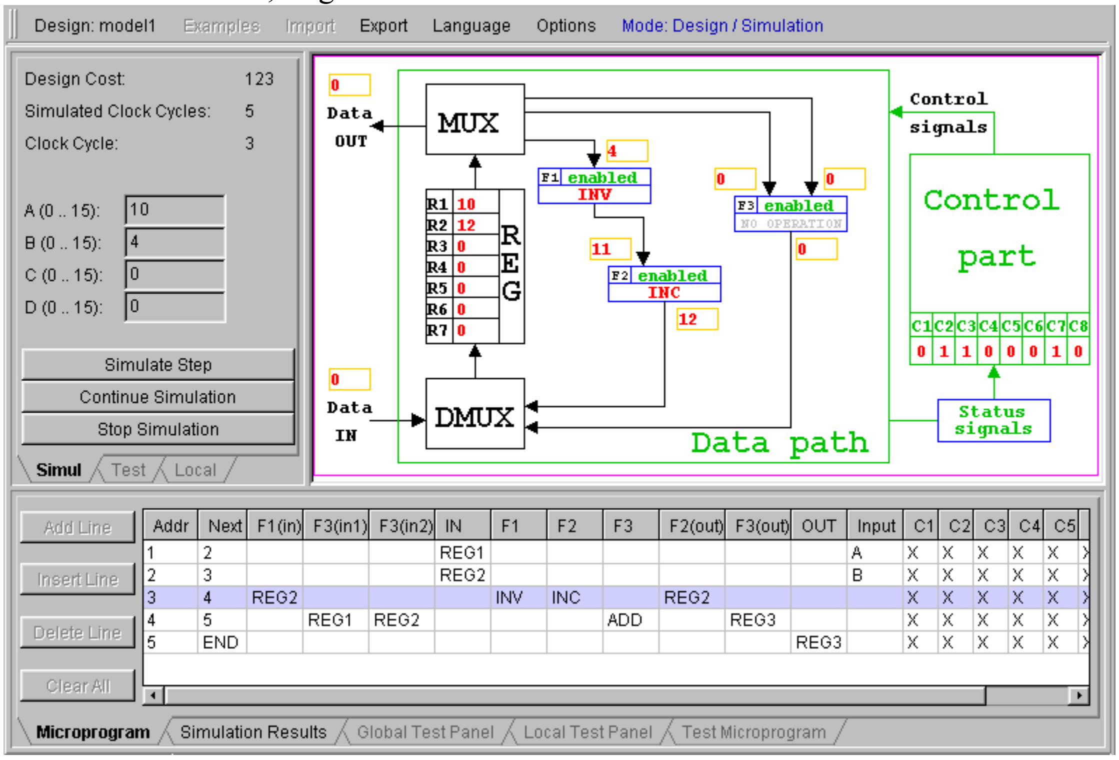
RT-Level teaching system illustrates many problems related to both RT-level control intensive digital design and test. This gives a possibility to teach all of them in a consecutive iterative approach. The range of problems includes:

- Design of datapath and a control part (microprogram) on RT- level
- Investigation of tradeoffs between speed & HW cost in the system
- RT- level simulation and validation
- gate-level deterministic test generation and functional testing
- fault simulation
- logic BIST, circular BIST, functional BIST, etc.
- design for testability

The teaching system interface consists of the following major parts:

- *Schematic View* - panel provides the schematic representation of a design. This panel allows user to define or change some properties of datapath of the design. Some components (functional units) can be enabled /disabled or their functionality can be changed. In the step-by-step simulation mode the results of the simulation are visually demonstrated on the *Schematic View* after each step of the simulation.
- *Microprogram tab-panel* is used to define control part of the system. During the simulation this panel shows, which part of the microprogram, is currently executed.
- *Simulation tab-panel* and *Test tab-panel* are used to simulate and test design correspondingly. The simulation can be carried out in different ways:
  1) *Step-by-step simulation mode*. In this case each row of the microprogram is executed separately and all the results of this execution (including state of registers and functional blocks, inputs and outputs, status signals, etc) can be viewed directly on *Schematics View* sub-panel. This mode of simulation is useful for illustrating how the design works or for debugging.
  2) *Test mode*: This mode is used to test the design repeatedly with some set of input data. User fills in the *Test data table* in the *Test tab-panel* with the data that will be used in testing. Then the design simulation can be executed for a particular row of test table or for all the rows at once. The results of the simulation or test are placed to *Simulation Results tab-panel.*
- *Simulation Results* tab-panel reflects the results of fault-free simulation.
- *Fault simulation* module provides fault simulation for the datapath and its units.
- *Global Test Panel* is used to provide fault coverage information as for the whole datapath as for each single unit under test.
- *Local Test Panel* provides means for manual local test patterns generation for a selected unit of datapath. It also provides the fault coverage for each unit as well as for the datapath as whole.

- *Test Microprogram* allows change the special test microprogram, which is used in Deterministic test, Logical and Circular BIST modes.



Design Cost: 123
Simulated Clock Cycles: 5
Clock Cycle: 3

A (0 .. 15): 10
B (0 .. 15): 4
C (0 .. 15): 0
D (0 .. 15): 0

| Addr | Next | F1(in) | F3(in1) | F3(in2) | IN | F1 | F2 | F3 | F2(out) | F3(out) | OUT | Input | C1 | C2 | C3 | C4 | C5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | | | | REG1 | | | | | | | A | X | X | X | X | X |
| 2 | 3 | | | | REG2 | | | | | | | B | X | X | X | X | X |
| 3 | 4 | REG2 | | | | INV | INC | | REG2 | | | | X | X | X | X | X |
| 4 | 5 | | REG1 | REG2 | | | | ADD | | REG3 | | | X | X | X | X | X |
| 5 | END | | | | | | | | | | REG3 | | X | X | X | X | X |

Figure 22. The interface of the teaching system

The teaching system has a flexible design, with several implemented RT-level system models (implemented RT model will be described in section 9), the latter allow developing different algorithms with different HW cost and speed. The teaching system has a built-in extendable collection of examples implementing different algorithms (multiplication, subtraction, etc). They help users to understand principles the system operation. For connecting the system to other applications as well as for providing users with a possibility to save the results of their work for further use, the applet has a data import/export capability. [4]

## Datapath description

| | | |
|---|---|---|
| **MUX** | $m_{ij}$: $B_j = R_i$ | i = 1,…,n – Register number; j = 0,1,2,4 – Bus number where $B_0$ is Data OUT Bus, $B_1$ is the Bus to F1 etc. |
| **DMUX** | $d_{ij}$: $R_j = B_i$ | i = 0,3,4 – Bus number where $B_0$ is Data IN Bus, $B_3$ is the Bus from F3 etc. j = 1,…,n – Register number |
| **F1 (F3)** | $f_{1j}$ ($f_{3j}$) | unary microoperations like: various shiftings, inverting, counting (+1, -1) etc. |
| **F2** | $f_{2j}$ | various binary microoperations (with 2 operands) |
| **F4** | $f_{4j}$ | various unary and binary microoperations (with 2 operands); there is a overlay between functions of F4 and the ones of F1, F2 and F3 to allow a parallelization of the |

Figure 23. Description of datapath functionality

Each functional unit (FU) of the datapath F1…Fn contains a number of microoperations (functions: unary and binary), which are labeled by corresponding control signals activating chosen function. The description of the datapath functionality in format "control signal: microoperation" is presented in Figure 23.

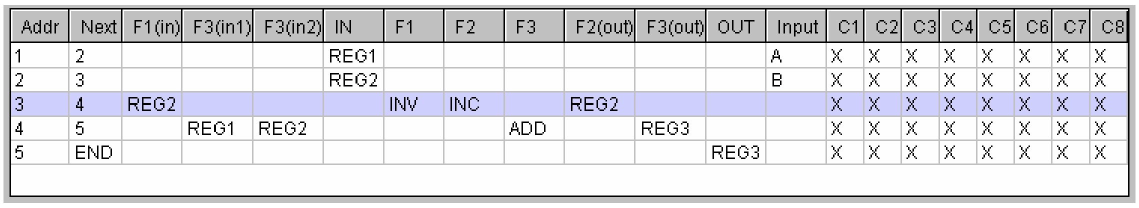
The user can select one or more microoperations for each unit of datapath when implementing his own algorithm (like subtraction, multiplication etc). Each microoperation has a gate-level implementation, and the number of gates determines its cost. All selected microoperations determine the final HW cost of the system.

Different architectures of the datapath can be chosen for implementation of the given algorithm, so the user can compare them considering either the cost or timing requirements. For example, the user can use only one functional unit, in order to carry out a single microoperation in one clock cycle. In this case the hardware cost is saved but the speed is low. There is another possibility, in which all functional units can be used, in a parallel or sequential mode, in order to carry out maximum number of microoperations during a single clock cycle, if the algorithm allows to. In this case the speed is higher than in the previous case.

The speed (the number of clock cycles) of the algorithm is measured by simulation. In its turn the simulation is supported by an RT-level model of the system as a whole and by gate-level models of each microoperation in each FU.

## Control part

The control part is a microprogrammed controller, which implements Mealy FSM (Final State Machine). The controller consists of a microprogram table and an interpreter. The microprogram is developed by the user to realize a given algorithm based on the selected resources of the datapath. The user fills in the rows of microprogram table, which contain information about the address of the current and the next microinstruction, MUX and DMUX configurations, Data IN values, selection of functions in FUs (F1 to Fn) at each microinstruction, and status signal configuration. In Figure 24 an example of subtraction algorithm of two operands A and B is presented. The result of subtraction is stored in REG3 and fed out to the data output.



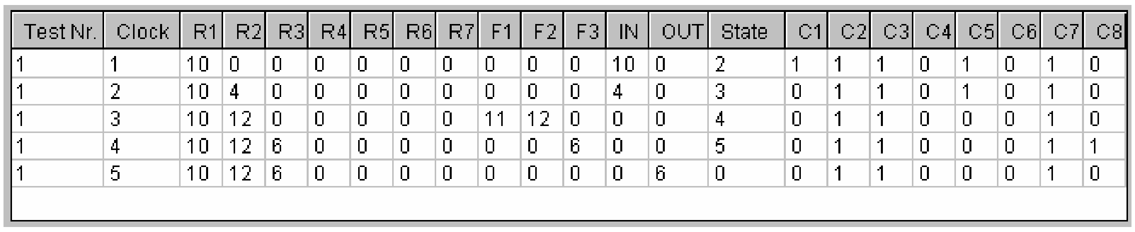
| Addr | Next | F1(in) | F3(in1) | F3(in2) | IN | F1 | F2 | F3 | F2(out) | F3(out) | OUT | Input | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | | | | REG1 | | | | | | | A | X | X | X | X | X | X | X | X |
| 2 | 3 | | | | REG2 | | | | | | | B | X | X | X | X | X | X | X | X |
| 3 | 4 | REG2 | | | | INV | INC | | REG2 | | | | X | X | X | X | X | X | X | X |
| 4 | 5 | | REG1 | REG2 | | | | ADD | | REG3 | | | X | X | X | X | X | X | X | X |
| 5 | END | | | | | | | | | | REG3 | | X | X | X | X | X | X | X | X |

Figure 24. Subtraction algorithm microprogram

The first two columns of the microprogram table represent the address of current microinstruction and the address of the next microinstruction correspondingly. The current microinstruction can be split into several rows in case if its operation depends on the set of conditions C. Then the only proper row will be selected. Columns F1(in). ..,Fn(in) correspond to MUX and indicate which register (REG1…REGn) will be multiplexed into which functional unit (F1, .., Fn). Registers where the input data from

Data IN (column “Input”) will be written are specified in column “IN”. The input data are the operands of the implemented algorithm. Columns F1 to Fn stand for a certain microoperation selected for a corresponding functional unit (F1 to Fn) in a certain clock cycle. The DMUX section is specified in columns F1 (out),...,Fn(out). It shows to which register the data from functional units $Fn_1$ and $Fn_n$ will be written. Column “OUT” indicates the register, which will be redirected to Data OUT. The last columns (C1, …, Cn) stand for conditions where the following values must be specified: 0, 1, X (don’t care).

The RT-level simulation is carried out at the higher level by using corresponding to functional units Java subroutines, which are activated according to condition values by the control signals in the order given in the microprogram table. The simulation data is stored in the Simulation Results subpanel, presented in Figure 25 below.



| Test Nr. | Clock | R1 | R2 | R3 | R4 | R5 | R6 | R7 | F1 | F2 | F3 | IN | OUT | State | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 2 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 2 | 10 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 3 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 3 | 10 | 12 | 0 | 0 | 0 | 0 | 0 | 11 | 12 | 0 | 0 | 0 | 4 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 4 | 10 | 12 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 5 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 5 | 10 | 12 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

Figure 25. The Simulation Result panel

Column “Test Nr.” defines the number of data group from the Test Data Table. The clock number “Clock” is specified in the next column. The simulation data is written in all other cells of the Simulation Results table at each clock cycle . This data reflects the states of all the registers, outputs of all the functional blocks, data input and output of the device, current states at each clock cycle and condition signals. The simulation data can be used by the student as a debugging info as well as for the improving the efficiency: the speed or the cost of the system.

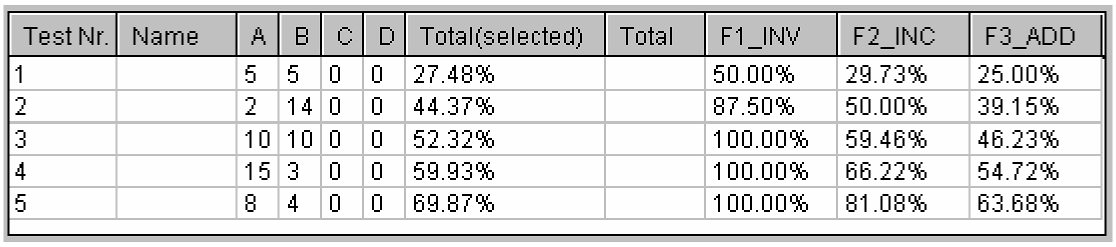
## Testing

The toolkit of the modern design and test engineer contains quite a few methods of testing of a SoC design. All of them have come from the earlier times and have been adopted for the new paradigm. The RT teaching system shows a variety of different modern testing techniques including functional and deterministic testing, a number of BIST solutions.

Prior to entering the test mode, the system under test must be designed and verified. The user can do it himself or use one of prepared examples. When the test mode is selected, the microprogram and the structure of the datapath are “frozen” and cannot be modified anymore. At the same time the user selects target microoperations of the data path for test generation and fault simulation. The fault simulation information is reflected (depending on a mode selected) at the Global Test Panel for the whole system and at the Local Test Panel for a single selected unit. In the following we describe the test modes in detail.

**Functional Testing**

In this mode the cheapest test technique is investigated, which does not require designing special test programs and embedding of special test structures into the system. The same unmodified microprogram and datapath are used instead.The required level of fault coverage must be achieved then only by a smart selection of input data. The fault simulation information is presented at the Global Test Panel. The input operands (A, B, C, D) are specified first. The same microprogram is used then repeatedly for fault simulation for all the input data. The fault coverage is calculated for each selected FU and for the whole system as well. The cumulative fault coverage (column "total") for each input vector is also provided in the Global Fault Coverage table, shown in Figure 26 presented below.



| Test Nr. | Name | A | B | C | D | Total(selected) | Total | F1_INV | F2_INC | F3_ADD |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | 5 | 5 | 0 | 0 | 27.48% | | 50.00% | 29.73% | 25.00% |
| 2 | | 2 | 14 | 0 | 0 | 44.37% | | 87.50% | 50.00% | 39.15% |
| 3 | | 10 | 10 | 0 | 0 | 52.32% | | 100.00% | 59.46% | 46.23% |
| 4 | | 15 | 3 | 0 | 0 | 59.93% | | 100.00% | 66.22% | 54.72% |
| 5 | | 8 | 4 | 0 | 0 | 69.87% | | 100.00% | 81.08% | 63.68% |

.

Figure 26. Global Fault coverage table

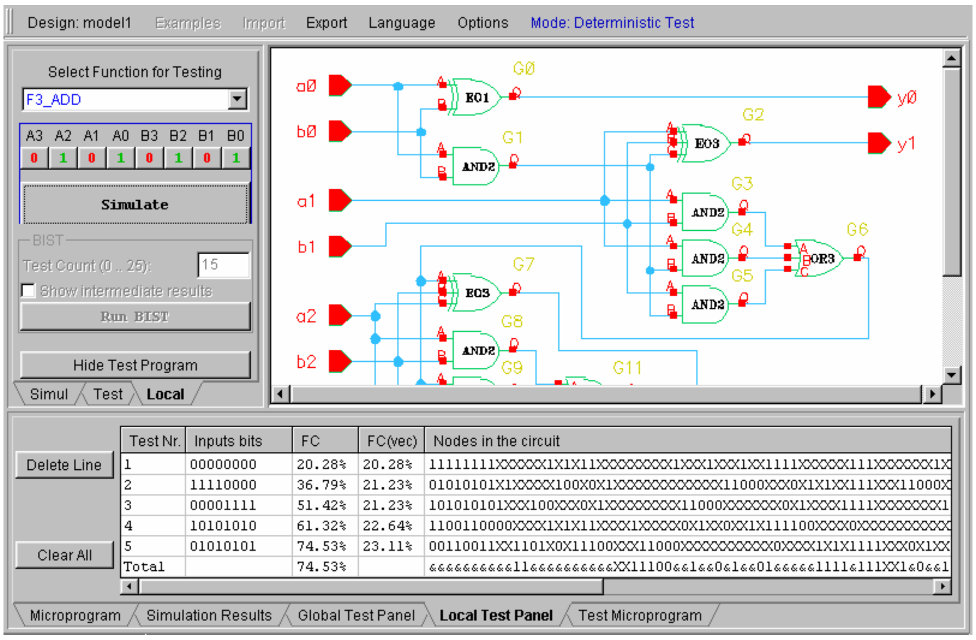
**Deterministic Test**

Deterministic Test mode is aimed at a gate-level test generation and fault simulation for each selected FU separately, see Figure 27. They are considered by the user in series and test vectors are generated. The simulation results are provided in the fault table at the Local Test Panel. For each vector the fault coverage (FC (vec)) is calculated and the information on tested nodes is given. The cumulative fault coverage (FC) is also shown for each simulation step.

The hierarchical RT-level fault simulation is also applied in order to evaluate the global fault coverage of those vectors for the datapath as a whole. For this purposes a test program is composed for each selected FU. The simulation data is reflected in the Global Test Panel in the same way as it is done in the Functional Test mode.

In order to help the user generating gate-level test vectors, the gate-level schematic of currently selected FU is displayed. The user selects a target fault and generates a test vector. After pressing the "Simulate" button this vector is fault simulated at the gate level and the results (local fault coverage) are added into the fault table. At the same time, the same vector is sent to RT-level hierarchical fault simulator in order to fill in the Global Test Panel.

The test microprogram, used for RT-level fault simulation must provide a good access to the selected FU. A simple version of such a program is generated automatically. It can be used as a template by a student in order to develop a more sophisticated test program if needed.



| Test Nr. | Inputs bits | FC | FC(vec) | Nodes in the circuit |
|---|---|---|---|---|
| 1 | 00000000 | 20.28% | 20.28% | 11111111XXXXX1X1X11XXXXXXXXX1XXX1XXX1XX1111XXXXX111XXXXXX1X |
| 2 | 11110000 | 36.79% | 21.23% | 01010101X1XXXXX100X0X1XXXXXXXXXXXXX11000XXX0X1X1XX111XXX11000X |
| 3 | 00001111 | 51.42% | 21.23% | 101010101XXX100XXX0X1XXXXXXXXX11000XXXXXXX0X1XXXX1111XXXXXXXX1 |
| 4 | 10101010 | 61.32% | 22.64% | 1100110000XXXX1X1X11XXXX1XXXXX0X1XX0XX1X111100XXXX0XXXXXXXXXXX |
| 5 | 01010101 | 74.53% | 23.11% | 00110011XX1101X0X11100XXX11000XXXXXXXXXXXX0XXXX1X1X1111XXX0X1XX |
| Total |  | 74.53% |  | &&&&&&&&&&&11&&&&&&&&&&&XX11100&&1&&0&1&&01&&&&&1111&111XX1&0&&1 |

Figure 27. Deterministic test pattern generation in Local Test Panel

## BIST mode

RT teaching system allows to provide testing with different BIST solutions, which are based on a scan-path technology (Figure 28), where the inputs and outputs of the combinational blocks in datapath are directly accessible by TPGs, SAs or TPG/SA (combined TPG and SA).

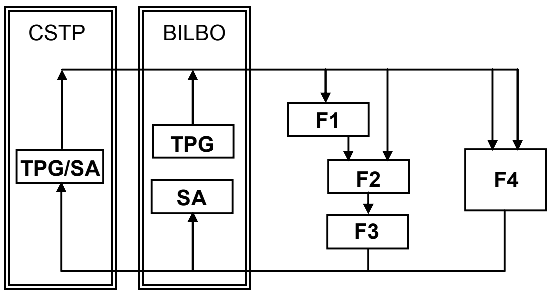
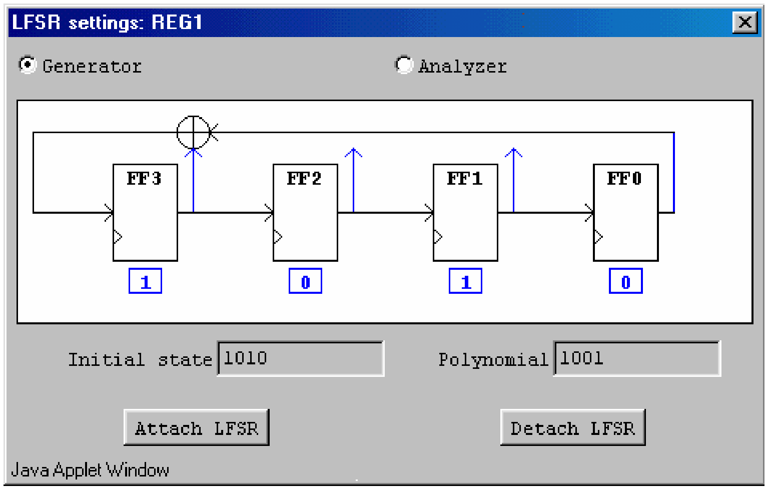


Figure 28. Scan-path design

Teaching system allows reconfiguration of internal registers in the BIST mode. Depending on the chosen BIST method some of them can perform functions of TPG, SA or TPG/SA. If the Logic BIST (BILBO) method is to be evaluated, the TPG and SA functions must be separated and located in different registers. On the contrary, in Circular BIST (CSTP) both TPG and SA are situated in the same register. In the both modes it is possible to configure the TPG on-line from the interactive graphical panel, see Figure 29.



[✓ Generator] Analyzer

Initial state 1010

Polynomial 1001

Figure 29. Interactive LFSR configuration graphical panel

When the configuration is completed, the gate-level and the hierarchical fault simulation are performed and the results are displayed in the way similar to the one used in Functional and Deterministic test modes.

There is another BIST mode, called Functional BIST. This mode has very much in common to Functional Testing. The only difference between the two modes is that in the former one there is possibility to insert SAs at any arbitrary point within the data path. In this way we increase the observability of the system, since each such SA is capable of collecting data at each clock compressing it into an observable signature.

## REFERENCES

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

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

[3] DANIEL. D GAJSKI, NIKIL D. DUTT, ALLEN C-H WU, *High-Level synthesis Introduction to Chip and System Design,* Kluwer Academic Publishers, 1994.

[4] S. DEVADZE, A. JUTMAN, A. SUDNITSON, R. UBAR, H.-D. WUTTKE *"Teaching Digital RT-Level Self-Test using a Java Applet"* 20th IEEE Conference NORCHIP'2002, Copenhagen, Denmark, 2002, p.322-328.

[5] JANUSZ RAJSKI, JERZY TYSZER. *Arithmetic Built-In Self-Test for embedded systems.* New Jersey, 1998.

[6] JOHN L.HENNESSY, DAVID A. PATTERSON. *Computer Organization and Design*. San Francisco, California, 1997.

[7] JAAN RAIK. *Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams*. Doctor graduate thesis. Tallinn, 2001.

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