An Expert System for Self-Testable Hardware Design

by

Kwanghyun Kim

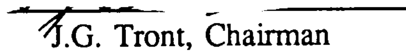
Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

APPROVED:



J.G. Tront, Chairman

J.R. Armstrong

E.A. Brown

D.S. Ha

S.F. Midkiff

May, 1989

Blacksburg, Virginia

An Expert System for Self-Testable Hardware Design

by

Kwanghyun Kim

Committee Chairman: Dr. Joseph G. Tront
Electrical Engineering

(ABSTRACT)

BIDES (A BIST Design Expert System) is an expert system for incorporating BIST into a digital circuit described with VHDL. BIDES modifies a circuit to produce a self-testable circuit by inserting BIST hardware such as pseudorandom pattern generators and signature analysis registers. In inserting BIST hardware, BIDES not only makes a circuit self-testable, but also incorporates the appropriate type of BIST structure so that a set of user-specified constraints on hardware overhead and testing time can be satisfied. This flexibility comes from the formulation of the BIST design problem as a search problem. A satisfactory BIST structure is explored through an iterative process of evaluation and regeneration of BIST structure. The process of regeneration is performed by a problem solving technique called hierarchical planning. In order to apply a hierarchical planning technique, we introduce an abstraction hierarchy in BIST design. Using the abstraction hierarchy, the knowledge of the BIST design process is represented with several operators defined on the abstraction levels. This type of knowledge representation in conjunction with hierarchical planning led to an easy implementation of the system and results in an easily modifiable system.

In this dissertation, we also study a BIST scheme called cascade testing. In cascade testing, a signature analysis register is used concurrently as a test pattern generator in order to reduce the overall testing time by improving testing parallelism. The characteristics of the patterns generated by the signature analysis register are investigated through analysis as well as experiments. It is shown that the patterns generated by signature analysis registers are rarely repeated when the number of patterns generated is relatively small compared to the number of all possible patterns. It is also shown that the patterns generated by signature analysis registers are almost random. Therefore, signature analysis registers can be used effectively as pseudorandom pattern generators. The practicality of cascade testing is investigated by fault simulation experiments using an example circuit.

Acknowledgements

First and foremost I would like to thank to Dr. Joseph G, Tront, my adviser, for being so supportive during this long haul. He has been so patient and always encouraging in spite of my poor performance. He has provided fine guidance insuring quality thesis and not detracting from the student's creative process. I believe that our interactions were extremely beneficial. A special thank you goes to Dr. Dong Sam Ha who made me realize the importance of publications. His kind and enthusiastic guidance has improved my writing skills greatly. I have especially enjoyed our discussions, which has been both productive and enjoyable. I would also like to thank to Dr. Ezra A. Brown who was willing to work hard in order for students to learn more. The process of logical thinking I have learned under his instruction has been solid basis for pursuing the degree. I also wish to thank Drs. James R. Armstrong and Scott F. Midkiff for serving on my committee.

A special appreciation is directed to my friend, Dr. Hyun-Taek Chang, who almost forced me to join this challenging field of computer engineering. He has saved words of praise to make me not be satisfied with little success. Instead, he has pushed me for the bigger success through careful criticism and suggestions. I am also very grateful to Dr. Bo Hyung Cho and for their mental support. Whenever I faced a mental barrier, they were willing to spend their time to save me from the deep.

In Korea, my family has encouraged me from a distance to do things they do not understand. I am deeply indebted to my parents for their limitless support to their son,

who has been very unfaithful to them for the past six years. I am also very grateful to my brother and sister who have taken care of everything that I was supposed to do. I wish to express my deep appreciation to my wife who has sacrificed everything for her husband's completion of study. Without my family, I could not have completed my degree.

Table of Contents

Chapter 1 Introduction 1

Chapter 2 Background 6

2.1 Built-In Self-Testing 6

2.1.1 Input Pattern Generation 9

2.1.2 Test Response Evaluation 10

2.2 BIST Implementation Mechanism 11

2.3 Previous Research on Automated DFT 14

Chapter 3 System Strategy 22

3.1 Problem Identification 22

3.2 BIST Design as a Search 26

3.3 Overview of the Design Process 27

3.4 Abstraction 30

3.5 Knowledge Representation 32

Chapter 4 Design Analysis 35

4.1 VHDL Modeling Style for BIST Design 35

4.2 Prolog Description of Hardware 42

4.3 Internal Representation of the Design Structure 44

Chapter 5 Test Structure Building 47

5.1 Initial State Generation (Phase 1) 47

5.2 Initial State Generation (Phase 2) 54

Chapter 6 Evaluation of BIST Implementation 61

6.1 Evaluation of Testing Time 61

6.2 Evaluation of Hardware Overhead 64

6.2.1 Built-In Logic Block Observer 66

6.2.2 Signature Analysis Register 70
6.2.3 Pseudorandom Pattern Generation 70
6.2.4 Scan Filp-flop 72
6.2.5 Total Hardware Overhead 72

Chapter 7 Regeneration of BIST Structure 75
7.1 Reducing Hardware Overhead 75
7.2 Reducing Testing Time 83
7.3 Scan-Path Organization 88
7.4 Control Signal Distribution 89

Chapter 8 Study on Cascade Testing 92
8.1 Background 92
8.2 Characteristics of Patterns Generated by SAR 95
8.2.1 Randomness 99
8.2.2 Effectiveness 101
8.3 Experimental Results 107
8.3.1 Randomness 107
8.3.2 Effectiveness 111
8.4 A Case Study 117

Chapter 9 Conclusions 123
9.1 BIDES 123
9.1.1 Contribution 123
9.1.2 Future Directions 124
9.2 Study on Cascade Testing 125

Bibliography 128

Appendix A: Prolog Description of Hardware 132

Appendix B: Syntax for Frame Representation of Objects 135
B.1 Test Structure Frame 135
B.2 Circuit Frame 137

B.3 Input Port Frame 137
B.4 Output Port Frame 138
B.5 Register Frame 139

Appendix C: Restrictions on VHDL Modeling 140

Appendix D: Examples 142
D.1 Example 1 142
D.2 Example 2 148
D.3 Example 3 154
D.4 Example 4 160
D.5 Example 5 165

Appendix E: Derivation of $P_m(k)$ in 8.2.2 174

Vita 175

Chapter 1
Introduction

The increasing complexity of Very Large Scale Integrated (VLSI) circuits has made the testing of the circuits more difficult. Among the reasons for this increase in testing difficulty are decreased accessibility of internal nodes due to the higher gate to pin ratios and new failure modes introduced by the new technology. One general approach to addressing this testing problem is embodied in a collection of techniques known as Design For Testability (DFT) [1].

The DFT techniques are divided into two categories [2]. The first category is that of ad hoc techniques for solving the testing problem. These techniques solve a problem for a given design and are not generally applicable to all designs. This is contrasted with the second category of structured approaches. These techniques are generally applicable and usually involve a set of design rules by which designs are implemented. Built-In Self-Testing (BIST) is one of the structured DFT techniques, providing for test pattern generation and test response evaluation to occur within the logic itself [3][4]. DFT techniques can alleviate the testing problem, but penalties are also incurred due to the set of design rules that must be obeyed. Major penalties are the additional silicon area and design cost. DFT can be introduced at all phases of the design process, but it should be considered early in the design process in order to reduce the penalties incurred.

There are two main ways to take testability into account while developing computer design-aids: (1) automated design tools which directly take into account testability constraints, (2) tools which verify compliance with DFT methodologies of already existing chips. Both cases describe particular problem aspects of the general design problem, i.e., the creation of an implementation starting from an abstract specification by means of intelligence operation.

A significant improvement in automated synthesis and verification has been produced by the introduction of knowledge-based expert systems, i.e., programs that mimic human behavior in problem-solving [5]. Such tools were originally developed by Artificial Intelligence (AI) researchers and applied to various domains, ranging from chemistry to medicine. Recent years have witnessed an increasing number of knowledge-based systems in Computer-Aided Design (CAD), as well as in the field of Computer-Aided Testing (CAT). There are many good reasons that predict the success of these AI CAD/CAT tools.

1. In general, the nature of the design problem is not yet well-understood and changes too rapidly for solutions to be encoded in algorithms.

2. Human experts are available for knowledge-transfer; typical problems take humans from several minutes to several hours to solve, thus they can be tackled by current technology.

3. Expert systems are very flexible and allow the tool makers to add, modify and validate new knowledge with relatively little effort.

Expert systems also have their drawbacks; human expertise is often incomplete and difficult to encode in a machine usable form. Also, programming tools supporting expert system implementation are very far from being efficient. Hence, up to now only a few systems have had a real impact on designers.

This dissertation presents a CAD tool called BIDES (BIST Design Expert System) that enables a designer to easily incorporate BIST into original IC designs. BIDES was developed using the knowledge-based expert system approach. Incorporation of BIST requires a high degree of expertise. Given that the designer will be completely engrossed in developing the functionality of a circuit, it would be advantageous to relieve him/her of some of the responsibility of incorporating testing into the design. BIDES can alleviate the need for the designer to be a Design For Testability (DFT) expert. Rather, the designer need only become familiar with the methodology and the tool, and be able to make trade-off decisions similar to those made in the original system design. This is the primary motivation of the research.

As will be reviewed, the process of BIST design can be characterized as the creation of control logic and testing resources. There are several ways to provide testing resources to a given design, each having different characteristics. Trade-offs also exist among BIST structures. Hence, BIST design must be performed according to the objectives of the designer. A CAD tool for incorporating BIST must be capable of finding a BIST implementation which meets the design objectives. The tool must be able to explore possible BIST implementations and evaluate them compared with the design objectives. This is the major objective pursued in developing BIDES.

The secondary goal pursued is to make an easily modifiable system. As technology develops, new testability problems may arise. Since BIST knowledge is dynamic, it is important that the knowledge in the tool be observable so that the tool can be modified easily. In order to make an easily modifiable system, an attempt is made to represent the BIST design knowledge abstractly by defining an abstraction hierarchy in BIST. The inclusion of abstract knowledge also makes the BIST design

represented in the system easier to understand. Moreover, this abstract knowledge will be the basis for the explanation capability of the system which can be implemented in the future.

In this dissertation, we also study a BIST scheme called cascade testing [6]. The cascade testing scheme has been used in testing real circuits [7][8] because it offers the advantage of short testing time. However, there has been no theoretical study for cascade testing. We investigate the characteristics of cascade testing analytically. Then, the result of this analysis will be reviewed through the experiments.

In Chapter 2, we discuss the background of the research. We first briefly review BIST techniques and their implementation mechanisms. Next is a review of the previous research on automated DFT.

In Chapter 3, we first identify the problem involved with automated BIST incorporation. Then, the problem solving strategy adopted in BIDES is discussed. The incorporation of BIST is defined as a search process which consists of an iterative process of evaluation and regeneration. Next, an overview of the design process based on the search strategy is described. The tasks necessary for automatied BIST incorporation performed by BIDES are specified. The last two sections of the chapter discuss the knowledge representation techniques adopted in BIDES. In order to provide the flexibility to allow for knowledge modification, an abstraction hierarchy in BIST design is defined. Using the defined hierarchy, knowledge about BIST design is represented with procedures on each abstraction level. The actual BIST incorporation is performed using the technique called hierarchical planning.

From Chapter 4 through Chapter 7, we describe each step in the design process in detail. In Chapter 4, we discuss structure extraction from a VHDL description of an original design, which is an input to BIDES. The VHDL description is translated into

a Prolog description which is processed by BIDES. We discuss an appropriate VHDL modeling style for BIST design. The Prolog description of the hardware is also described.

As the initial step of the search process for finding a right type of BIST structure, Chapter 5 describes the test structure building step; the initial BIST structures are incorporated into the design. In Chapter 6, the methods for evaluation of a BIST implementation are described. A BIST implementation is evaluated in terms of hardware overhead and testing time. In Chapter 7, we describe the final step in the search process, the regeneration of BIST implementation. If the current BIST implementation cannot satisfy the design objectives, then a new BIST implementation is generated by modifying the current implementation. We will discuss how hierarchical planning techniques can be used in BIST design. In Chapter 7, the final two steps of BIST design, scan-path organization and control signal distribution, are also described.

In Chapter 8, we study a BIST scheme in which a signature analyzer is used as a test pattern generator. In order to use the signature analyzer as a test pattern generator, the patterns generated by the signature analyzer should have similar characteristics with the patterns generated by a typical test pattern generator. We analytically study two aspects, randomness and repetition of patterns. The analytical results are compared with experimental results. In Chapter 9, the dissertation is summarized.

Chapter 2

Background

2.1 Built-In Self-Testing

As the complexity and size of VLSI circuits has increased, so has the difficulty of testing these circuits. An early solution to this testing problem was to increase the accessibility of the internal storage elements by using latch scanning arrangements. The techniques known as LSSD [9], Scan/Set [10], Random Access Scan [11] are included in this category. Since these techniques depend on conventional testing strategy, i.e., test patterns are stored in test equipment and applied through pins of a chip, the burdens of test pattern generation and fault grading still remain. The increasing complexity of modern VLSI makes these operations very time consuming. Moreover, testing at-speed is very difficult since each test pattern must be inserted through long shift register chain(s).

As an alternative to the scan-path techniques, Built-In Self Testing (BIST) has been proposed [3][4]. BIST has received much attention over the past few years since it can be used to prevent unrestrained growth of testing costs. The basic principle of BIST is to generate test patterns and evaluate test responses inside a chip. A schematic diagram of the basic BIST configuration is shown in Figure 1. As shown in Figure 1, test patterns are generated by the internal Test Pattern Generator (TPG) and applied to the Circuit Under Test (CUT). Testing is initiated by activating a small set of control pins on the chip. Then, test responses of the CUT are evaluated by the Test Response Evaluator (TRE) and the result is observed through one or two output pins.

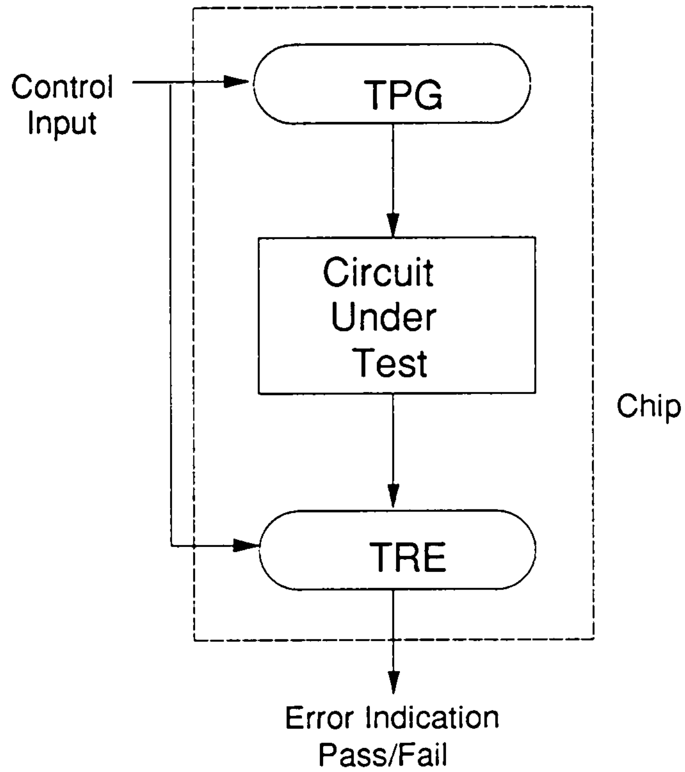


Figure 1. BIST of a circuit.

According to the observed test responses, the test result is determined to be either pass or fail. Since test pattern generation and test response evaluation are performed inside a chip, BIST has some significant advantages over conventional testing as enumerated below:

- Test vectors can be generated automatically inside the chip at high speed.
- Neither test patterns nor correct test responses need to be transferred or stored external to the chip.
- Testing can be processed at much higher speed than by using VLSI hardware testers.
- The need for expensive test hardware is eliminated.
- Testing may be more easily applied in the field.

Although BIST has the potential to alleviate some of the problems of conventional testing or scan-path techniques, there are some requirements for making a circuit self-testable. Since test patterns are to be generated internally, they must be easy to generate and generic to all different types of circuits. Another requirement for BIST is the need for modification of the CUT. For example, all sequential circuits need to be modified so that testing of a sequential circuit can be transformed to the testing of a combinational circuit. Regular structured components such as ROMs, RAMs, and PLAs also need this type of modification. Therefore, the following penalties will be incurred in BIST:

- Additional silicon area and pins are consumed by the testing logic.
- Operational performance may be degraded because of the added BIST circuitry.
- Additional design time and cost are necessary to incorporate BIST.

2.1.1 Input Pattern Generation

There are two general test pattern generation approaches for BIST. One of the methods, pseudorandom testing, uses a set of pseudorandomly generated bit patterns as test patterns. The other approach, exhaustive testing, uses all possible input combinations as test patterns. Random testing has the advantage of being applicable to sequential as well as combinational circuits. Random patterns are usually generated by means of a simple circuit called a Linear Feedback Shift Register (LFSR) [12]. The circuit is known as a Pseudorandom Pattern Generator (PPG) in BIST. The major difficulty in using random patterns is in determining the required test sequence length to achieve a desired fault coverage. One straightforward method for determining fault coverage is to use fault simulation. However, for large complex circuits, simulation time becomes fairly long. The other drawback encountered in using fault simulation is the need for developing a fault model. This model depends on the fabrication technology. A number of analytical methods [13][14] have been proposed for estimating fault coverage, but extensive computation is required to obtain high accuracy.

Exhaustive testing has many desirable attributes including the following: no need for fault modeling, no fault simulation is performed, pattern generation is straightforward, simple hardware, and some timing faults can be detected. However, in order to use the exhaustive testing technique, it is necessary that the logic is able to be partitioned into small enough pieces so that exhaustive testing can be performed on sufficiently small modules. Otherwise, the number of exhaustive test patterns that must be applied to a fairly large block of logic can cause the testing time to become prohibitive. Unfortunately, there is no known good partitioning algorithm. Therefore, this approach will remain impractical until a practical partitioning algorithm is found.

2.1.2 Test Response Evaluation

In conventional testing, every test output response is compared to the correct response. If this scheme is adopted in BIST, too much on-chip memory space is required to store the expected responses. In order to reduce the volume of output data, several output compaction techniques have been proposed. These include signature analysis [15], transition counting [16], and syndrome testing [17]. Transition counting, a good technique for board-level testing, has not received serious consideration for use in BIST since better methods have been recently developed. In syndrome testing, output response compaction is done by simply counting the number of 1's output when an exhaustive test pattern is applied. If all single stuck-at faults can be detected by this scheme, a circuit is called *syndrome testable*. However, the scheme still relies on the exhaustive testing approach and as previously mentioned, exhaustive testing remains impractical.

Signature analysis is the most popular compaction technique for BIST. Compaction is done by using an LFSR with its inputs fed by the output responses of the CUT. The term "signature" describes the contents of the LFSR after all the test response patterns of the CUT are shifted into the LFSR. There is one important problem in signature analysis called *aliasing*. It is possible for a fault to cause a CUT to generate an output bit sequence that produces the same final LFSR contents as does the fault-free circuit. In this case, the fault is not detectable. Generation of output sequences that exclude the possibility of aliasing depends upon the structure of the LFSR used. There are two methods used in the signature analysis of multiple output circuits. One method is called *serial signature analysis*. This method uses a multiplexer to direct each of the outputs to the LFSR in turn. The input test patterns must be applied m times for m-output circuits. The other scheme is *parallel signature*

analysis which compacts m circuit outputs in parallel using an m-bit parallel code checker. A parallel signature analyzer, which is known as a Multiple Input Signature Register (MISR), is faster but requires more circuitry than the serial signature analyzer. In this dissertation, MISR will be implied when a Signature Analysis Register (SAR) is specified.

2.2 BIST Implementation Mechanism

As described in the previous section, exhaustive testing is impractical. Thus, only implementation techniques based on random pattern testing will be reviewed here. Several BIST architectures based on random pattern testing have been proposed in the literature [18][19][20][21][22][23]. All of these architectures use pseudorandom patterns as test patterns while they use signature analysis techniques to compact the test responses. Another common technique employed in these architectures is that of using a scan-path. The two reasons for employing scan-path techniques are (1) registers can be tested independently using the scan-path, (2) internally compacted output responses can be observed at the chip output through the scan-path. Incorporation of a scan-path technique transforms the testing of a sequential circuit into the testing of a combinational circuit since all the memory elements are controllable and observable.

Although the same general strategies are used in most BIST architectures, there are some differences in the hardware structures used. BIST structures can be classified into two groups according to the implementation of the self-testing hardware. The approach used in the first group is called *centralized* S^3 (Self-Testing using Scan-path and Signature Analysis) [19]. In this approach, centralized self-testing hardware is responsible for testing all of the modules in a design. Test patterns are usually applied through the scan-path. The second approach is a *distributed* S^3 technique in which

many PPGs and SARs are distributed throughout a design. In this approach, generated test patterns are applied to combinational logic circuits through the system data path, not through a scan-path. Since testing is performed using the system data paths, testing resources such as PPGs and SARs are provided by converting existing system registers into multi-functional testing resources. The advantage of the centralized approach over the distributed approach is the reduction in hardware overhead. However, the overall testing time is longer than that of the distributed approach. Due to the lengthier testing time of the centralized approach, the distributed approach is preferred over the centralized approach. In the following paragraphs, therefore, we will review the BIST structures based on the distributed approach.

Suppose that the distributed approach is applied to the circuit configuration in Figure 2. Assuming the use of scan-path techniques, two Combinational Logic Blocks (CLBs), CLB A and CLB B, need to be tested. In order to test CLB A the register R1 needs to be converted into a PPG in order to generated pseudorandom patterns. Also, R2 needs to be converted into an SAR in order to collect the signature of CLB A. For the testing of CLB B, R2 needs to be converted into a PPG and R3 needs to be converted into an SAR to collect the signature of CLB B. Notice that R2 is converted to act as an SAR to test CLB A and a PPG to test CLB B. This approach is called *Built-In Logic Block Observer* (BILBO) [18]. A BILBO is a multi-functional register which has four functional modes, PPG, SAR, Shift Register and Parallel Load modes. All three registers, R1, R2 and R3, need to be connected as a shift register chain in order to scan-in the initial pattern and scan-out the collected signature. Since there is a conflict in the roll of R2, the two CLBs cannot be tested in parallel.

One method for testing the two CLBs in parallel is to use the SAR of CLB A as the test pattern generator of CLB B. Registers R2 and R3 need to be modified to

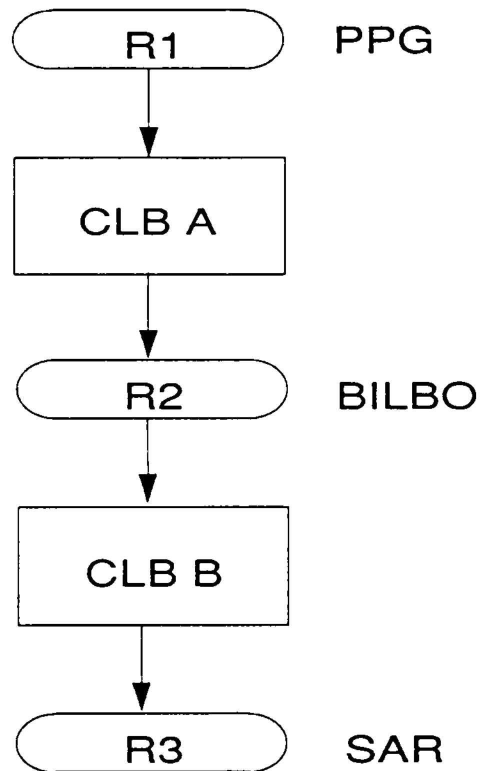


Figure 2. BILBO architecture for two cascaded modules.

act only as SAR for CLB A and CLB B, respectively as shown in Figure 3. During testing, the PPG modified from R1 applies pseudorandom patterns to CLB A, and the test responses of CLB A are collected by the SAR R2. At the same time, the contents of R2 serve as test patterns for CLB B. The signatures of CLB A and CLB B will be scanned out through the scan-path to evaluate the test. This testing technique is called *cascade testing* [6].

The other circuit configuration in which an SAR can be used as a PPG is shown in Figure 4. If a register is used as input as well as output register as shown in Figure 4, the register can be used PPG or SAR, but not both. In order to make this kind of circuit configuration pseudo-random testable we need an extra register as shown in Figure 5. However, if the existing register is used as a PPG and SAR at the same time, the extra register can be eliminated as shown in Figure 6. In this dissertation, this testing scheme is called *feedback testing*. The major problem with feedback testing is the patterns generated by SAR. The number of distinct patterns generated by the SAR is less than that of a PPG for the same number of clocks. However, it is reported that testing time is not much longer when compared to the pseudorandom testing [7][8].

Due to the multi-mode operations of registers in BIST, control logic must also be provided for proper configuration of registers in each mode. For example, a BILBO register has two signals to control its four functional modes. Therefore, the BIST design process can be characterized as the creation of testing resources and control logic.

2.3 Previous Research on Automated DFT

Considering the tremendous ongoing effort in developing design automation tools, it seems to be possible to automate the entire design process in the near future.

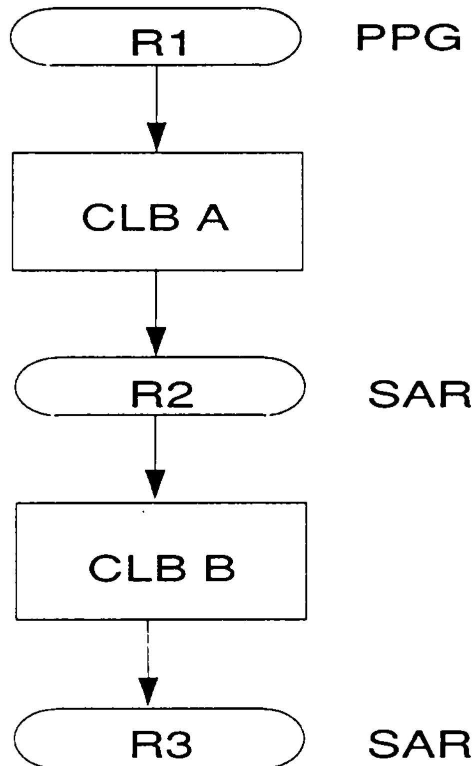


Figure 3. Parallel testing of two cascaded modules (cascade testing).

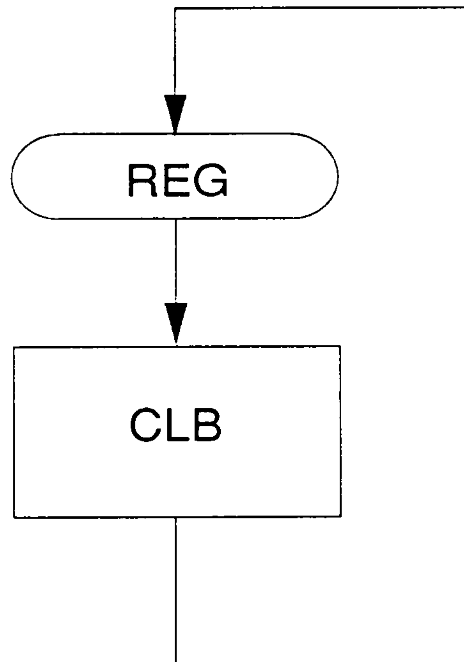


Figure 4. Circuit configuration with feedback.

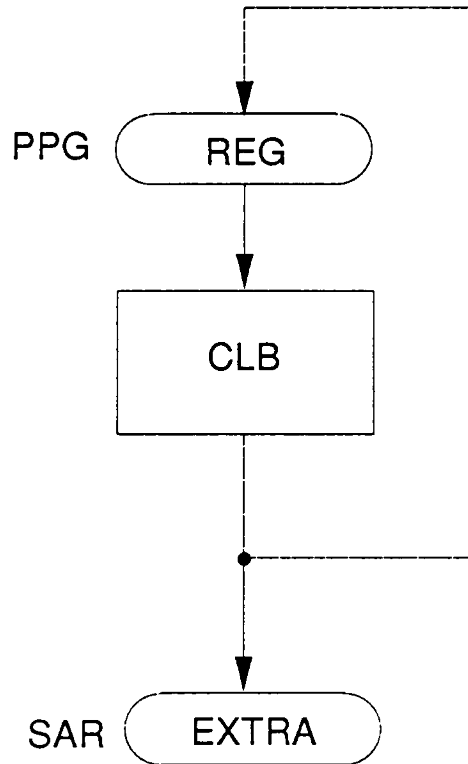


Figure 5. Pseudorandom testing of a circuit with feedback.

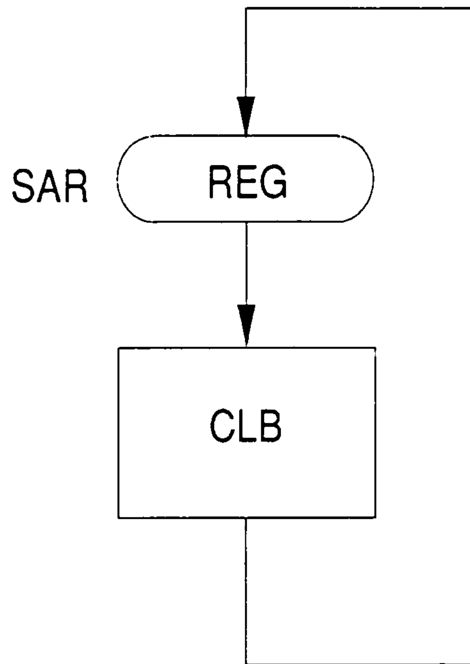


Figure 6. Feedback testing.

Then, in order not to slow the design cycle, the process of DFT also needs to be automated. This is the major motivation for various work on automated DFT. As mentioned in Chapter 1, DFT tools can be categorized into two groups, one for verification of DFT rules and the other for the automatic insertion of testability features. In this section, previous research on both issues will be reviewed, but the tools for automatic DFT synthesis will be emphasized since BIDES belongs to the latter category.

The pioneering effort in the field of computer-aided DFT systems was the work done by Horstmann [24]. Based on the LSSD DFT methodology, a system was implemented using Prolog which can detect violations in testability rules for LSSD. The system could then modify the circuit to remove the DFT violations.

Agrawal *et al.* [25] developed a design automation tool called TITUS which checks circuits for design rule compliance, implements testability hardware, and generates tests. The system includes the handling of on-chip memory and chip layout techniques to minimize hardware overhead and performance penalties.

ESTA (Expert System for Testability Automation) [26] is another expert system to verify compliance of existing designs with DFT methodologies, such as LSSD and BILBO. ESTA checks the DFT rules using a hierarchical description of a design. The techniques is called hierarchical verification. ESTA's knowledge-base integrates production rules and frames, yielding a hybrid system.

The above three tools were developed for DFT rule verification and operates on the logic and transistor level. In the following paragraphs, the tools for DFT synthesis are introduced. Fung *et al.* developed an Automatic DFT (ADFT) [27] system that works in conjunction with the Silc silicon compiler being developed at GTE Labora-

tories [28]. Testability evaluation is performed by using controllability/observability methods and a testable-by-construction approach is followed in order to synthesize blocks of logic.

Samad *et al.* developed DEFT [29], an expert system capable of accepting existing designs created by IBM's MVISA Design Automation System [30] and adding to them DFT features. The representation scheme is Prolog-based and the system has a demon mechanism allowing it to use LISP functions. Frames are used to represent structural knowledge about DFT methodologies. A sophiscated explanation mechanism [31] is put at the user's disposal.

ADFT and DEFT systems are both related to BIDES, but input description of the systems is at the logic level description. Because of progress in high level, i.e., behavioral synthesis systems, there is a need to insert DFT features at higher levels. Since high level synthesis significantly reduces the design cycle at the system level, delaying the DFT/BIST insertion to a lower level may require costly design modification. This is the major reason for inserting DFT features at the register transfer level. The CAD tools most relevant to BIDES are the Testable Design Expert System (TDES) [32] and the system developed by Jones and Baker [33] since these systems consider the BIST insertion at the register transfer level. BIDES also uses the register transfer level description of a design as an input.

The TDES was developed by Abdadir *et al.* to work with the ADAM system being developed at the University of Southern California [34]. The input to the TDES consists of register transfer level description of a circuit. The TDES knowledge-base is organized into Testable Design Methodologies (TDMs) that are implemented using frames. Each TDM consists of several templates such as description of the structure to which the methodology is applicable, the hardware that must be added to make the structure testable, and a test plan describing the steps that must be taken to apply the

test. TDES first identifies the "kernels" (a partition of a circuit) to which TDMs can be applied and then matches each structure in the TDM frame with a structure in the circuit. If more than one TDM is applicable to a kernel, TDM measures are estimated so as to choose between them. TDM measures are estimates of the costs associated with using a TDM. TDES has been implemented in LISP.

Another knowledge-based system for high-level BIST design is proposed by Jones and Baker. This system relies heavily on the BILBO-based testing. The basic strategy is to insert an appropriate number of BILBOs so that the given constraints on hardware overhead and testing time are satisfied. The system presents a range of test plans which meet the constraints imposed by the designer and achieve a balance between hardware overhead and test time. From the set of possible test plans, the designer can select a suitable one that best matches the original design considerations.

Several useful concepts are proposed in TDES such as application of a suitable DFT technique to each part of a design and frame representation of TDMs. However, this system lacks the capability of exploring alternative implementations when a DFT technique is applied. Our previous work [35] also lacks this capability. As will be shown in the example circuit in the next chapter, there could be many BIST structures based on the BILBO technique only. The system proposed by Jones and Baker has this capability, but a limitation of the system is that the knowledge for searching alternatives is not represented abstractly. Abstraction through defining level of hierarchies and operations on each level make it easier to understand the procedure for finding a right type of BIST scheme and the BIST scheme itself. This will lead to an easily modifiable system. In BIDES, it is attempted to formalize the procedure by defining level of abstractions and operations.

Chapter 3

System Strategy

As reviewed in the previous chapter, there are several BIST techniques. They differ in strategy for input pattern generation and response evaluation. Among the various strategies, random pattern testing with signature analysis has been used most widely because of its practicality and wide applicability. For this reason, BIDES considers the implementation of BIST based only on random pattern testing. Other testing strategies, e.g, exhaustive testing, are not considered in BIDES. Another restriction placed on BIDES is the testing of special structures such as ROMs, RAMs, and PLAs. Self-testing of these special structures requires modification of the circuit itself at the lower level. We need dedicated CAD tools for automated design of self-testable PLAs and RAMs. BIDES is developed for BIST of random logic, rather than special structures. Thus, testing of special structures is not taken into account in BIDES.

With the above restrictions, in this chapter, we will first identify the problem involved with automated incorporation of BIST. Next, the problem solving strategy will be discussed. Then, an overview of the design process based on the strategy will be followed. In the last two sections, we will describe the knowledge representation technique used in BIDES.

3.1 Problem Identification

As reviewed in Section 2.2, the BIST design process can be characterized as the creation of control logic and the set of testing resources. However, for a given design,

there are usually several ways to provide testing resources. Consider the implementation of BIST on a fragment of a design shown in Figure 7. Assuming the use of a scan-path technique for testing registers, testing resources need to be provided to two CLBs, CLB A and CLB B. In implementing the PPG for CLB A, there are two candidate registers, R2 and R3, since transparent paths, i.e., a path through which data can be transferred without any change, exist from R2 and R3 to the input of CLB A. Similarly, R3 and R4 can be used as a PPG for CLB B. There are three choices, R1, R2, and R4, for the SAR of CLB B and all four registers can be used as an SAR for CLB A.

Among the several ways of allocating testing resources, three possible schemes are shown in Table 1. In Scheme 1, all four registers need to be converted into testing resources, but R3 does not need to be modified in Scheme 2. Since extra circuitry must be added to convert a normal register into a testing resource, the hardware overhead of Scheme 1 is greater than that of Scheme 2. However, Scheme 1 has the advantage over Scheme 2 in terms of testing speed. In Scheme 2, R1 is used as SAR for both CLBs. Thus, two CLBs cannot be tested in parallel due to the resource sharing conflict on R1, and the testing time will be $TT_A + TT_B$. TT_i denotes the required number of pseudorandom patterns for testing CLB i. On the other hand, there is no sharing of resources (path and/or testing resource) in Scheme 1 and the parallel testing of two CLBs is possible. Therefore, the testing time for Scheme 1 is $max(TT_A , TT_B)$, which is lower than that of Scheme 2.

If we want further reduce the hardware overhead, Scheme 3 can be used. As shown, it is necessary to modify only two registers, R2 and R4. In Scheme 3, test patterns for CLB B are <u>not</u> pseudorandom patterns. R4 is used as the SAR for compressing test responses of CLB B. At the same time, the contents of the SAR (R4) are

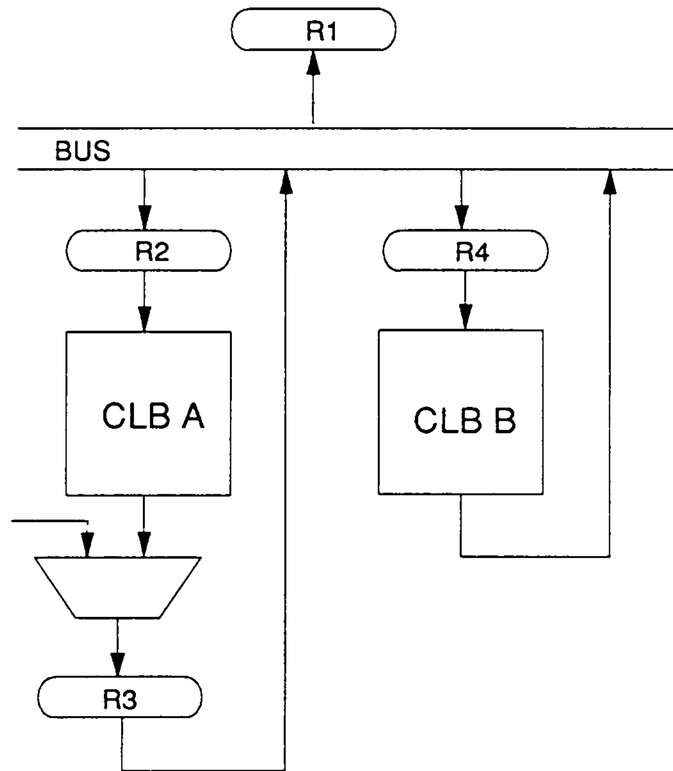


Figure 7. A fragment of a design.

Table 1. Three Schemes for Testing Two CLBs in the Design in Figure 1.

	Scheme 1		Scheme 2		Scheme 3	
	TPG	SAR	TPG	SAR	TPG	SAR
CLB A	R2	R3	R2	R1	R2	R4
CLB B	R4	R1	R4	R1	R4	R4
Modified Registers	R1, R2 R3, R4		R1, R2, R4		R2, R4	
Hardware Overhead	High		Medium		Low	
Testing Time	$\max(TT_A, TT_B)$		$TT_A + TT_B$		$TT_A + \alpha * TT_B$	

(TT_i denotes the required number of pseudorandom patterns for testing CLBi. $\alpha > 1.0$)

used as test patterns for CLB B. This is the feedback testing scheme introduced earlier. However, Scheme 3 has the longest testing time among the three testing resource allocation schemes because test patterns are generated by the SAR, not PPG. The required number of test patterns generated by the SAR (R4) is larger than TT_B since patterns generated by the SAR are repeated while all the patterns generated by the PPG are distinct [6].

The observation which can be made in the above example is that there is no one best BIST implementation for a given design. There exists a trade-off between hardware overhead and testing time. Therefore, BIST design must be performed according to the design objectives of the designer rather than using a fixed BIST template. The CAD tool for incorporating BIST must be capable of finding a BIST implementation which meets the design objectives. The tool must be able to explore possible BIST implementations and evaluate them compared with the design objectives. This is the main goal to be pursued in developing BIDES.

3.2 BIST Design as a Search

As shown in the example in the previous section, there are several possible solutions for allocating testing resources. Each solution has different characteristics. The problem is how to find a solution, i.e., a testing resource allocation, that meets the design objectives in terms of hardware overhead and testing time. Thus, the problem of testing resource allocation can be defined as a search through a space of solution states.

The general problem solving strategy for the search, especially in design problems, is the process of evaluation and regeneration [36]. For a given state, a new state

is generated by a form of transformation and is evaluated compared to the goal state. By the iterative process of evaluation and regeneration, an initial state will eventually be evolved to a goal state. The regeneration and evaluation of various BIST structures requires a high degree of expertise in BIST. This is the primary motivation for employing the expert system approach.

There is an important issue in solving the problem of testing resource allocation using search. That is the concept of a state. In BIDES, a state is defined as a complete solution, i.e., a BIST implementation with which the entire design can be tested. For example, Scheme 1, Scheme 2 and Scheme 3 are all complete solutions for the design in Figure 7. A state could be defined as a partial solution; BIST implementation for a part of a design. In order to define a state as a partial solution, there must be a reliable evaluator with which it can be judged whether a partial solution is a part of satisfactory complete solution or not [37]. In testing resource allocation, it is very difficult to make such an evaluator since the overall testing time cannot be evaluated for a part of BIST implementation. Therefore, all the states generated during the search process including an initial state are complete solutions in BIDES.

Considering that all the states are complete solutions, any state in the solution space can be a goal state. If a state, i.e., a BIST implementation can satisfy the objectives of a designer on testing time and hardware overhead, that state is a goal state.

3.3 Overview of the Design Process

The overall design process, based on the strategy mentioned above, is shown in Figure 8. The first step of the process is the initial design analysis. In this step, the

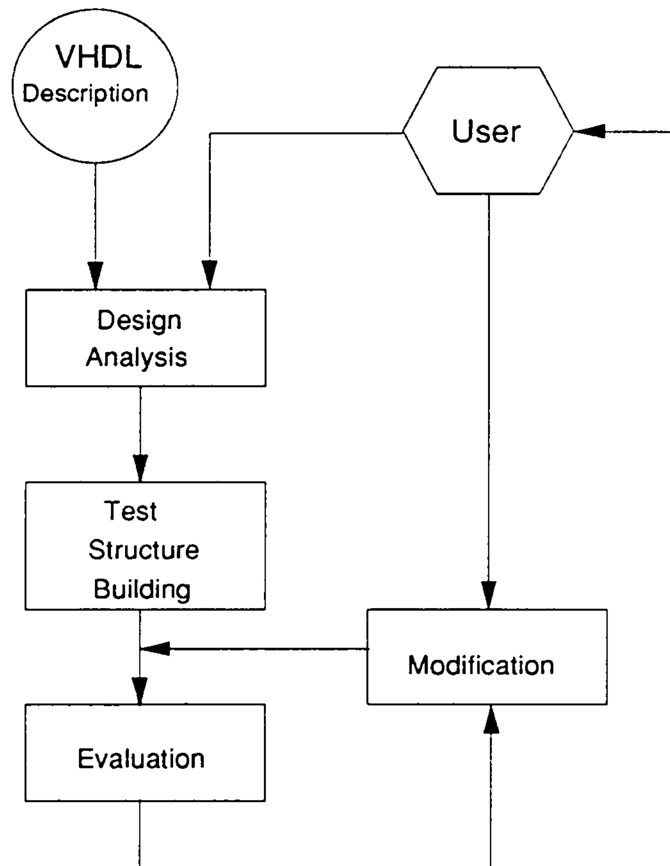


Figure 8. Overall design process.

structure of a design is extracted from the VHDL description of a design and stored in a database using a Prolog description of the hardware. An appropriate VHDL modeling style for BIST design and the Prolog description of the hardware are described in the next chapter. Next, circuits to be tested and usable testing resources are identified. Once the identification process is completed, the next step is *test structure* building. A *test structure* is a circuit configuration which is self-testable. It consists of testing resources and a CUT. A more detailed definition of a test structure will be given in the next section. The main purpose of this step is the generation of the initial state, i.e, all the CUTs identified in the design analysis step are made pseudorandom testable by allocating testing resources to each CUT.

The final step is an iterative process of evaluation and modification of the current state. The current state is evaluated first. Then, if the current state cannot satisfy the designer's objectives in terms of hardware overhead and testing time, an alternative BIST implementation, which is closer to a goal state, will be generated by modifying the current BIST implementation. Modification is performed by allocating other testing resources or implementing variations of the pseudorandom testing scheme. The iterative process of evaluation and regeneration will be continued until the designer is satisfied or no other new solution can be found. After all of the testing resources are allocated, control signals for testing resources will be distributed. Finally, the scan-path will be organized.

In summary, the tasks of BIST design considered by BIDES are as follows:

- translation of the VHDL description into a Prolog description,
- allocation of testing resources,
- evaluation of testing time and hardware overhead for a given testing resource allocation,

- Scan-path organization, and
- control signal distribution.

3.4 Abstraction

Abstraction is the structuring of objects for simplifying the model of the world. By abstraction, reasoning for problem solving can be performed more easily using simpler objects and operations. We can observe the immediate need for abstraction in testing time evaluation. As shown in the example in Section 3.1, the overall testing time is determined based on the parallel testing of CLBs. Thus, the possibility of parallel testing must be examined for each pair of CLBs to evaluate testing time; sharing of testing resources (registers) and data transfer paths in testing mode should be checked. We need information on registers and data paths rather than on the CLB itself. From this observation, we define an object called a "test structure" which consists of a CLB and testing resources associated with it. Then, the parallel testing of two CLBs can be checked by defining an operator called "parallel testing" which examines the resource sharing between two test structures as shown below:

$$\text{parallel_testing}(TS_i, TS_j, Ans)$$

where TS_k denotes the test structure associated with the CLB k and "Ans" is a value returned by the operator, which is "yes" or "no".

By introducing the concept of test structure, an abstraction hierarchy for the BIST implementation can be defined as shown in Figure 9. A design consists of several test structures, but the test structures are not disjoint. A component, e.g., a register, can be included in more than one test structure. There are two kinds of test structures, *micro* test structure and *macro* test structure. The micro test structure is built for testing a

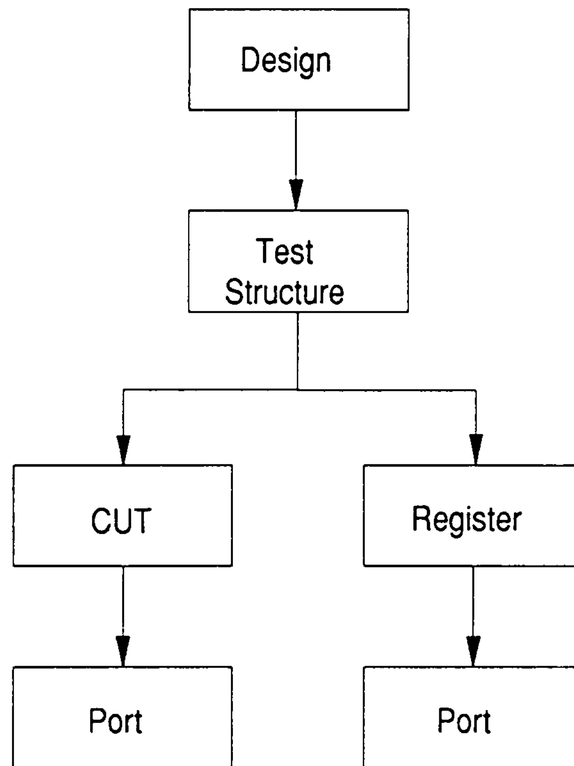


Figure 9. Abstraction hierarchy for BIST design.

single CLB. The macro test structure is produced when a testing scheme is used to test more than one CLB. For example, a macro test structure is produced when two CLBs are tested simultaneously by the cascade testing scheme. The objects in the next level of the hierarchy are the CUT and the register. The lowest level object is the port. Note that testing resource allocation is performed eventually on the ports.

An example of a test structure for CLB A in Figure 7 is described below using a knowledge representation technique called *frames* [38]:

```
frame(test_struct_1).
value(test_struct_1,cut,CLB_A).
value(test_struct_1,test_scheme,pseudorandom).
value(test_sturct_1,test_time,550).
value(test_struct_1,ppg,[[IN,R2]]).
value(test_struct_1,sar,[[OUT,R3]]).
```

As shown above, each value predicate contains the name of the test structure along with a parameter identifier and the value of the parameter. A test structure consists of a CUT and the allocated testing resources. It also has information about the testing scheme used and testing time of the CUT. The representation of objects in other levels of the hierarchy will be described in detail in the next chapter. This abstraction hierarchy defined above plays a vital role in representing the knowledge about the BIST design process. In the next section, knowledge representation techniques will be described.

3.5 Knowledge Representation

In BIDES, the knowledge about the BIST design process is represented with *heuristic knowledge* and *procedural knowledge*. Heuristic knowledge is a kind of

advice on what to do given a particular set of circumstances. In the BIST design process, there are many cases wherein a decision needs to be made. In BIDES, decisions are made using the heuristics. Heuristic knowledge is represented using if-then production rules.

Non-heuristic procedural knowledge [39] consists of plans or sequences of actions that must be taken to achieve problem-solving goals. Procedural knowledge is represented through several levels of abstraction. A plan can be described at a high-level as a conjunction of abstract goals; each of these goals can then be refined into subgoals at lower levels of abstraction. *Hierarchical planning* [40] is a problem solving technique based on procedural knowledge. In BIDES, a hierarchical planning technique is used as a framework in conjunction with the abstraction hierarchy defined in the previous section. Transformation in the search process and the second part of test structure building (test structure building is performed in two phases) are performed in three steps, *plan generation*, *plan analysis*, and *plan execution*, based on hierarchical planning.

In plan generation, a goal based on the difference between the current state and the goal state is first posted. These kinds of goals are "reduce hardware overhead" or "reduce testing time". Then, the options and requirements for achieving the posted goal is generated in the form of an AND/OR goal tree. The goal at the root of the AND/OR goal tree is represented in abstract terms and the goal is expanded into conjunctive subgoals until the subgoals can be solved using the primitive operators. An example of a primitive operator is "Allocate the register R to the port P". In the plan analysis step, the success of the plan is checked first. If there is a way to achieve the goal, the best plan based on the current situation, considering interaction between subgoals, is

chosen using the plan decision knowledge. Otherwise, another plan is generated. Plan decision knowledge is also represented using production rules. A task is finally completed by executing a chosen plan.

By defining a test structure as a circuit configuration which is self-testable, the eventual goal is to build test structures for the CUTs in a design. Then, if a BIST implementation for the design needs to be modified to satisfy the given design objectives in the search process, the test structures need to be modified. Since each test structure is represented as an explicit object, the modification of a test structure can be posted as a goal. In order to modify test structures, we need to change the relationships between a CUT and testing resources, i.e., registers. Therefore, the goal at the test structure level can be expanded into the goals at the CUT and register levels. Finally, the necessary modification to the ports can be specified using the primitive operators. This is the lowest level in the hierarchy. In this way, it is easy to represent knowledge about the BIST design process and follow the procedure taken by BIDES. This type of knowledge representation leads to an easily modifiable CAD system since knowledge can be added or changed by defining necessary operators at each level.

Chapter 4

Design Analysis

As the first step of the design process, BIDES takes a VHDL description of a hardware design and translates it into a Prolog description. Then, the structure of the design is extracted and stored in the system database with the internal presentation format. In this chapter, an appropriate VHDL modeling style and syntax for the Prolog description are described. We also describe the internal representation of the structure of a design based on the abstraction hierarchy.

4.1 VHDL Modeling Style for BIST Design

VHSIC Hardware Description Language (VHDL) was developed to serve as a standard hardware description language with which design data can be communicated between engineers. Another objective in the development of VHDL is the integration of design automation tools through the use of a common language [41]. The level of description possible in VHDL ranges from system level to gate level. It also allows for modeling a design in many different styles, e.g., behavioral modeling and structural modeling. In order to use VHDL for BIST design it is necessary to define a modeling style and level of abstraction appropriate for accomplishing the tasks of BIST design. From the tasks listed for BIST design, we can reach the following conclusions about an appropriate VHDL modeling style.

1. The basic components in BIST, i.e., the PPG and the SAR, can be configured from the existing system registers. If there is a path from a register to a CLB, that register can be used as a PPG. Likewise, if there is a path from a CLB to a

register, that register can be used as an SAR. This means that we need information about the interconnection between registers and CLBs. Therefore, the most appropriate modeling style is structural which specifies interconnections between components at the register transfer level. At this level, designs are described in terms of register, bus, multiplexer, ALU, PLA, etc.

2. When random pattern testing is used on a CLB, knowledge of the detailed structure of a CLB is not necessary in order to add BIST circuitry. Therefore, a behavioral model can be used for describing each component. It does not mean that other modeling styles, e.g., data flow modeling or structural modeling, cannot be used. However, insertion of BIST hardware should be considered before the low level design description is frozen. Thus, a behavioral model is the most appropriate modeling style for describing each component.

3. A classification or type for each system component must be recognizable from the VHDL description. In order to allocate testing resources to CLBs in a design, type of each system component is necessary. Since type of component cannot be recognized using interconnection of components, the VHDL description must have the information on type of components.

Using this modeling style, part of TMS 32010 digital signal processor [42], shown in Figure 10, is described in VHDL. Figure 11 shows the VHDL description. As shown, the description is a typical structural decomposition, i.e., component declaration followed by component instantiation [43]. There are two important conventions followed in the description. First, in order to recognize the types of components in the system, prefixes are attached to the name of each component. For example, the prefix, "REG" is used for naming all of the register components. Second, the types of signals are divided into four classes; data, control, bus, and clock signal. Signals are also

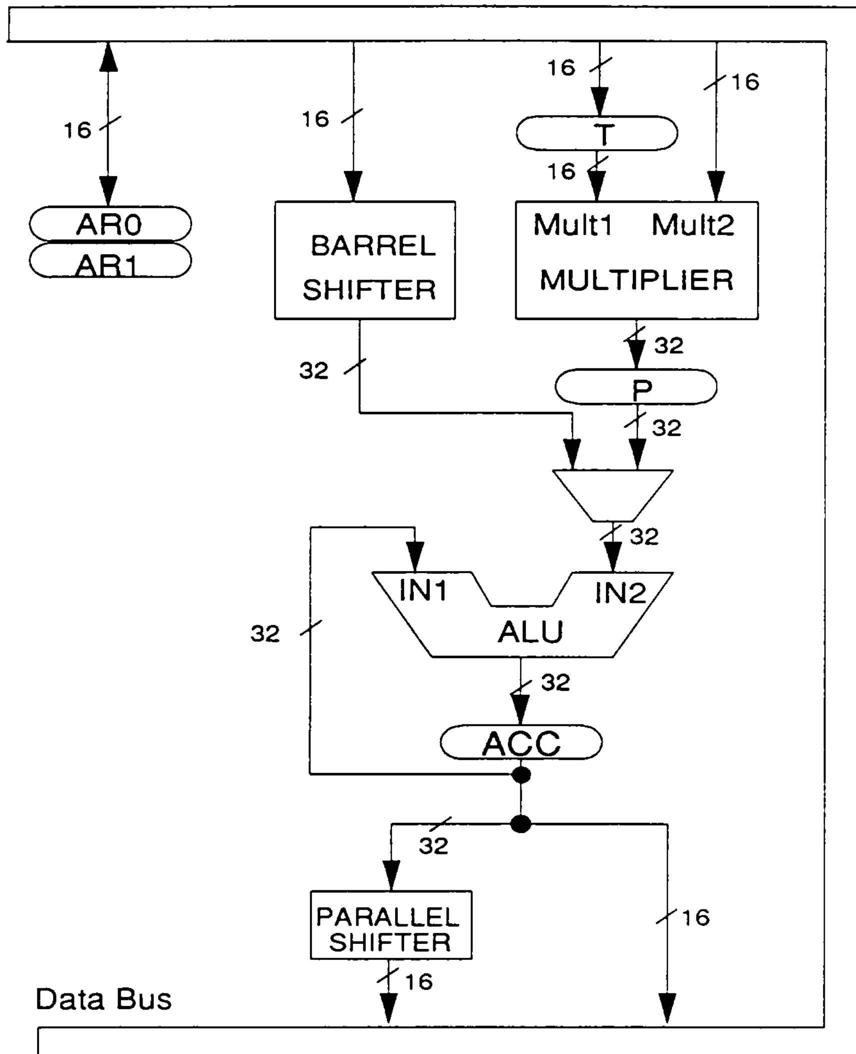


Figure 10. Part of TMS 32010 data path (adapted from [42]).

```
---------------- Interface of TMS 32010 ------------

package tri_bus is
    type        MV is ('0', '1', 'Z');

    type        MV_VECTOR is array (integer range <>) of MV;
    function    tri_resolve (sources: MV_VECTOR) return MV;
    type        tri_vector is array (integer range <>) of tri_resolve MV;

    subtype     tristate is tri_vector (0 to 15);

    function    "AND"  (X,Y : tri_vector) return tri_vector;
    function    "OR"   (X,Y : tri_vector) return tri_vector;
    function    "NOT"  (X : tri_vector) return tri_vector;
end tri_bus;

use DATAPATH.all;
entity  TMS_DATAPATH is
        port( CTL_ALU     :  in   BIT_VECTOR  (0 to 3);
              CTL_BA_SH   :  in   BIT_VECTOR  (0 to 3);
              CTL_PA_SH   :  in   BIT_VECTOR  (0 to 3);
              CTL_MUX2    :  in   BIT;
              CLK         :  in   BIT;
              BUS_DATA    :  inout tristate);
end TMS_DATAPATH;
```

Figure 11. A structural modeling for the design in Figure 10.

```
------------------------ Architecture of TMS 320 -----------------------

architecture STRUCTURAL of TMS_DATAPATH is
    component  CLB_MULT
               port  ( Mult1       :in     BIT_VECTOR;
                       Mult2       :in     BIT_VECTOR:
                       Output      :out    BIT_VECTOR);

    component  CLB_ALU
               port  ( in1         :in     BIT_VECTOR;
                       in2         :in     BIT_VECTOR;
                       Cntl        :in     BIT_VECTOR;
                       Output      :out    BIT_VECTOR);

    component  CLB_BARR_SHIFT
               port  ( Input       :in     BIT_VECTOR;
                       Cntl        :in     BIT_VECTOR;
                       Output      :out    BIT_VECTOR);

    component  C_PARA_SHIFT
               port  ( Input       :in     BIT_VECTOR;
                       Cntl        :in     BIT_VECTOR;
                       Output      :out    BIT_VECTOR);

    component  MUX_32
               port  ( in1         :in     BIT_VECTOR;
                       in2         :in     BIT_VECTOR;
                       Select      :in     BIT;
                       Output      :out    BIT_VECTOR);

    component  REG_32
               port  ( D     :in     BIT_VECTOR;
                       CP    :in     BIT;
                       Q     :out    BIT_VECTOR);
```

Figure 11. A structural modeling for the design in Figure 10 (cont'd).

```
component  REG_ACC_32
           port  ( D       :in     BIT_VECTOR;
                   CP      :in     BIT;
                   Lhalf   :out    BIT_VECTOR;
                   Q1      :out    BIT_VECTOR;
                   Q2      :out    BIT_VECTOR);

component  REG_16
           port  ( D       :in     BIT_VECTOR;
                   CP      :in     BIT;
                   Q       :out    BIT_VECTOR);


------------ Local Signal Declaration -----------------------

signal      MULT_OUT      :   BIT_VECTOR   (0 to 31);
signal      TREG_OUT      :   BIT_VECTOR   (0 to 15);
signal      PREG_OUT      :   BIT_VECTOR   (0 to 31);
signal      BA_SH_OUT     :   BIT_VECTOR   (0 to 31);
signal      ALU_LFT_IN    :   BIT_VECTOR   (0 to 31);
signal      ALU_RGT_IN    :   BIT_VECTOR   (0 to 31);
signal      ALU_OUT       :   BIT_VECTOR   (0 to 31);
signal      PARA_SH_IN    :   BIT_VECTOR   (0 to 31);
```

Figure 11. A structural modeling for the design in Figure 10 (cont'd).

```
--------------- Component Instantiation Statements ---------------

begin

    MULT      : CLB_MULT
    port map  (TREG_OUT,BUS_DATA,MULT_OUT);

    ALU       : CLB_ALU
    port map  (ALU_LFT_IN,ALU_RGT_IN,CTL_ALU,ALU_OUT);

    BARR_SH   : CLB_BARR_SHIFT
    port map  (BUS_DATA,CTL_BA_SH,BA_SH_OUT);

    PARA_SH   : CLB_PARA_SHIFT
    port map  (PARA_SH_IN,CTL_PA_SH,BUS_DATA);

    MUX2      : MUX_32
    port map  (BA_SH_OUT,PREG_OUT,CTL_MUX2,ALU_RGT_IN);

    PREG      : REG_32
    port map  (MULT_OUT,CLK,PREG_OUT);

    ACC       : REG_ACC_32
    port map  (ALU_OUT,CLK,BUS_DATA,PARA_SH_IN,ALU_LFT_IN);

    AR0       : REG_16
    port map  (BUS_DATA,CLK,BUS_DATA);

    AR1       : REG_16
    port map  (BUS_DATA,CLK,BUS_DATA);

    TREG      : REG_T_16
    port map  (BUS_DATA,CLK,TREG_OUT);

end STRUCTURAL;
```

Figure 11. A structural modeling for the design in Figure 10 (cont'd).

classified using prefixes. The prefixes "CTL", "CLK", "BUS" are used for control signals, clock signals, and bus signals, respectively. All other signals are assumed to be data signals.

4.2 Prolog Description of Hardware

For BIST design, the structure of a design should be extracted from the VHDL description and stored in the system database. For this purpose, we use a Prolog description of the hardware since we need a representation which can be processed by an expert system. There are several methods for describing hardware using Prolog [44]. Among these, we decided to used the *extensional method* [24]. This method is chosen since it closely resembles the VHDL description. As in VHDL, the extensional method description expresses the connectivity between components from a component viewpoint. The Prolog description of a design consists of clauses, each describing the individual modules in a design. The following example shows the clause for describing the P register in the example circuit of Figure 10.

```
module (preg,reg,
        [inport(d,mult,output,data,bit_vector(0,31)),
         inport(cp,clk,clk,clock,bit)],
        [outport(q,mux2,in2,data,bit_vector(0,31))]).
```

As shown in the above clause, a component is described using the predicate "module". A clause for a module has four arguments: name, type, a list of input ports and a list of output ports. One element of the list of input and output ports is also a clause which has five arguments. For example, the clause shown below represents the first input port "d" of the module "preg" with the arguments having the following

meanings:

inport(d,mult,output,data,bit_vector(0,31))

- d: the name of the first input port of the module "preg".
- mult: the name of the module connected to the input port "d".
- output: the name of the port of the module "mult" connected to the port "d".
- data: the type of signal at port "d".
- bit_vector(0,31): the size or datapath width of port "d".

From the above example, it can be seen that a clause for a module has all of the connectivity, size, and type information about the module. In VHDL, the primary inputs and outputs are defined in the interface description. Those primary inputs and outputs are described as modules in the Prolog description. For example, a primary input port "ctl_alu" in the example circuit is described as

```
module (ctl_alu,primary_in,
        [],
        [outport(alu_ctl,alu,cntl,cntl,bit_vector(0,3))]).
```

Since there is no input to a primary input port, the list for the input port is null. The name of the output port is the same as the name of the module since there is only one port. A bus is also defined as a module since it can be considered as a module having inputs and outputs. The complete Prolog description of the design shown in Figure 10 is given in Appendix A. These examples provide some insights as to the contents and format of the Prolog description that can be extracted form the VHDL description.

4.3 Internal Representation of the Design Structure

From the Prolog description of the design, CUTs and testing resources are identified and represented using the internal format. Since the use of scan-paths is assumed, the CUTs that need to be identified are the CLBs in the design, and the testing resources are the existing system registers. As mentioned earlier, the testing of special structures such as memories and PLAs is not considered. In the example design, there are four CLBs (Barrel Shifter, Multiplier, ALU, Parallel Shifter), and five usable testing resources (AR0, AR1, T, P, ACC). The two types of modules, CLB and register, are represented by several entities in the abstraction hierarchy since knowledge about the BIST design is represented with operators defined on each level of the hierarchy. In describing each entity, a frame is again used. As an example, the frame representation of the Multiplier is shown below:

```
frame(circuit2).
value(circuit2,is_a,circuit).
value(circuit2,name,mult).
value(circuit2,no_of_pr_patterns,2645).
value(inputs,[inport4,inport3]).
value(outputs,[outport2]).
```

As shown above, a circuit frame is represented using the index which is internally generated. The index of the Multiplier is "circuit2". The "is_a" and "name" slot represent type and the original name of the circuit, respectively. The "no_of_pr_patterns" slot represents the number of pseudorandom patterns required to obtain a satisfactory fault coverage. The above example shows that 2645 pseudorandom patterns need to be applied to the Multiplier to obtain a satisfactory fault coverage. In BIDES, it is assumed that the designer has information on test length of each CLB. A circuit frame

also has slots for input and output ports. Each port of a CLB is again represented as a frame. The relation between CUTs and ports are represented with the slot "a_part_of" indicating that the port is a part of the CUT. The frame for the port Mult1 is shown below:

```
frame(inport3).
value(inport3,is_a,port).
value(inport3,name,mult1).
value(inport3,a_part_of,circuit2).
value(inport3,size,16).
```

Registers are also represented using frames. The frame representation of the register ACC is shown below:

```
frame(reg5).
value(reg5,is_a,register).
value(reg5,name,acc).
value(reg5,inport,d).
value(reg5,outport,q).
value(reg5,size,32).
```

The detailed syntax of the frames for a CLB, register, and the ports is described in Appendix B.

After construction of the frames for CLBs and registers, usable testing resources are identified for each port of the CUT. For example, AR0, AR1, T, and ACC are possible candidates to act as the PPG for the port Mult1 since transparent paths exist from the four registers to the port Mult1. The identified testing resources for the port are recorded under the slots "sources" or "sinks" of the port frame as shown below:

value(inport,sources,[reg2,reg3,reg4,reg5]).

If there is no available testing resource for a port, the value facet of the "source" or "sink" slot is given as "not_avail".

After all the usable testing resources for each port are identified, the testing resource allocation can be started. In the next chapter, the test structure building step, i.e., the initial testing resource allocation, will be described.

Chapter 5

Test Structure Building

As mentioned earlier, we consider a state in the solution space as a complete test resource allocation. The initial state must also be a complete test resource allocation, i.e, all of the CUTs in a design must have testing resources. Since a goal state will be evolved from an initial state, it is desirable to generate an initial state which requires few number of modifications. The basic testing strategy adopted by BIDES is pseudorandom testing. Thus, if all of the CUTs are made pseudorandom testable, the number of modifications necessary is expected to be reduced. In BIDES, test structure building, i.e., initial state generation, is performed in two steps called Phase 1 and Phase 2. In this chapter, each step will be described in detail using the example design shown in Figure 10.

5.1 Initial State Generation (Phase 1)

In order to make a CUT pseudorandom testable, a register must be allocated to each port of the CUT from among the registers identified as usable testing resources for the port. In allocating registers for pseudorandom testing, the following three constraints must not be violated.

1. The registers allocated to input ports must be distinct.
2. The registers allocated to the input ports cannot be allocated to the output ports

and vice versa.

3. Data paths, e.g., a bus, cannot be used in test pattern application and response collection at the same time.

If the first constraint is violated, the same test patterns will be applied to different input ports of the CUT. This would reduce the coverage provided by the PPG. The second constraint is for preventing the simultaneous use of the same register as a PPG as well as an SAR for the same circuit. Testing cannot be run in a continuous manner if the third constraint is violated. If this constraint is to be violated, it is necessary to perform some scheduling [45] for cooperative use of the shared data path.

Because of these three constraints, the problem of register allocation for pseudorandom testing can be defined as a *constraint satisfaction* problem. In order to solve this problem an algorithm based on *dependency-directed backtracking* [38] is devised and shown below.

Algorithm for initial testing resource allocation

1. Find all ports of a CUT which need testing resource allocation.

2. For all primary input ports

    allocate an external ppg;

    end for

   For all ports with no available resource

    record "no available resource";

    end for

3. Sort the ports with available resources in order so that the port with the least number of available resources can be put on the leftmost side.

4. If there is an unmarked port

    pick the leftmost port among the unmarked ports;

    mark this port as the current port;

  else

    stop;

5. Make a list which consists of all registers usable as a testing resource for the port.

6. For all registers already allocated to the ports of the CUT

    if a register is a member of the list of available resources for the

    marked port

      remove the register from the list of usable registers;

      post the port which used the register as a constraint;

      end if

    end for

7. If the list of usable registers is not empty

    select a register and allocate it to the port using the heuristic

    knowledge;

    mark the port;

    go to the step 4;

  else

    find the closest port on the right side which propagates the

    constraint which affects the allocation to the port;

    mark the current port as current backtrack;

    mark the found port as the current port;

    go to step 8;

  end if

8. Withdraw the register which is allocated to the port;

   remove the allocated register from the list of usable registers;

   if the list is empty

       if there is no constraint used

           go to the current backtrack port;

           record the current backtrack port as a failed port;

           go to the step 4;

       else

           reset the allocations for the ports placed on the left side of the current port;

           go to step 4;

       endif

   end if

Suppose that this algorithm is applied to implement the pseudorandom testing of the Multiplier which has two input ports, Mult1 and Mult2, and one output OUT. The first step is goal expansion. The top level goal

make a CUT pseudorandom testable (Multiplier)

will be expanded into the following three conjunctive subgoals on ports:

1. Allocate a register to Mult1.

2. Allocate a register to Mult2.

3. Allocate a register to OUT.

In the subgoal expansion process, ports such as primary inputs, primary outputs, and ports with no usable resource are ignored. For the primary input and output ports, the use of external PPG and SAR is assumed. Register allocation for a port with no resource will be performed in Phase 2.

The above subgoals interact with each other due to the constraints imposed. The result of the allocation will be different according to the order of subgoal execution. Therefore, we need to decide on the order of subgoal execution. The strategy "hard to satisfy first" is employed. For each port of a circuit, there could be several candidate registers that can be allocated. If priority is given to a port with the fewest number of candidate registers, the chance of violating the given constraints can be reduced. The port Mult1 has four candidate registers (AR0, AR1, T, ACC) to act as the PPG. The port Mult2 has three candidate registers (AR0, AR1, ACC). The register P is the only available register for the port OUT to act as the SAR. Therefore, the above three subgoals will be executed in the order of OUT, Mult2 and Mult1.

According to the order of subgoal execution, the next step is the register allocation. Allocation for a port starts by using the imposed constraints to eliminate unusable registers from the candidate register list. If there is still a usable register after elimination, allocation proceeds, otherwise backtracking is performed. When there is more than one usable register for a given port, a register is selected based on heuristics. Heuristics are developed to maximize the resource sharing, i.e, minimize the hardware overhead while the strict pseudorandom testing strategy is maintained.

For example, consider the allocation of a register to the port Mult2. Mult2 has three resources, AR0, AR1, and ACC. Assuming that the register P is already allocated to the port OUT, P cannot be used as the PPG for the port Mult2. However, P is not a usable resource for the port Mult2 anyway. Thus, the constraints do not affect anything in this case. The next step is register selection. First, the two registers, AR0 and AR1, are preferred over ACC because pure pseudorandom patterns cannot be generated by ACC. Since the size of ACC is 32-bits wide, the patterns generated by ACC can be pseudorandom patterns only for a 32-bit port. Then, we need to select a register

from the two registers, AR0 or AR1, since both registers can generate pure pseudo-random patterns for the port Mult2. In this case, a register is selected based on global resource sharing. If there is a register which is already allocated to act as a testing resource (PPG or SAR) for another CLB, then the register will be allocated. In that way, resource sharing can be maximized. If none of the registers is already used, a measure for potential resource sharing is used. It is called *Global Degree of Sharing (GDS)*. For example, the register AR0 can be used as a testing resource for four ports. Therefore, the GDS of AR0 is 4. However, since AR1 has the same GDS, a register from amongst these two will be selected arbitrarily. The above heuristics are implemented with *if-then* production rules. Some of them are shown below.

IF current port is an input port,
current context is "get usable resource" step,
there is more than one usable resource,
there is a resource whose size is not equal to the size of the port.

THEN remove the register from the set of usable resources.
change the current context to select step.

IF current port is an input port,
current context is "select step",
there is more than one usable resource,
there is a used resource,

THEN allocate the register to the port as the PPG.

IF current port is an input port,
current context is "select step",
there is more than one usable resource,
there is no used resource,
there is more than one resource with the maximum GDS,

THEN allocate a register with the maximum GDS.

Whenever a subgoal has failed, backtracking is performed based on dependency. That is, backing up to the last subgoal that could have affected the failure. If all the usable testing resources of a port are already allocated to other ports of the CUT, the port does not have any usable testing resources due to the imposed constraints. In this case, a backtrack is performed to the last port that produced the constraint posted on the current port. After backtracking, another choice, i.e., an allocation of another register, is searched for. If there is another choice, then allocation resumes from the backtracked port. Otherwise, backtracking continues until the allocation can be resumed or the leftmost port is reached. If backtracking does not resolve the goal failure, the cause of the failure is recorded and allocation proceeds ahead. The cause of a goal failure will be repaired in Phase 2.

Execution of the algorithm successfully built the test structure for pseudorandom testing of the Multiplier. The register AR0, T, and P are allocated to the port Mult1, Mult2 and OUT, respectively. Note that T is the only register which can be allocated to the port Mult1 since all of the other resources (AR0, AR1 and ACC) can apply patterns only through the Data Bus. Since the Data Bus must be used to apply patterns to the port Mult2, the Data Bus cannot be used to supply test patterns to the port

Mult1. That is, T must be used as the PPG for Mult1 in order to have continuous pseudorandom testing. The result of the allocation is recorded in the test structure as shown below.

```
frame(test_struct_2).
value(test_struct_2,cut,Mult).
value(test_struct_2,test_scheme,pseudorandom).
value(test_struct_2,test_time,2645).
value(test_struct_2,ppg,[[Mult1,T],[Mult2,AR0]]).
value(test_struct_2,sar,[[OUT,P]]).
```

In the above test structure, the real names of ports and registers are used for explanation purposes. In the real database, the index of the port and register is used instead of the name itself.

Modification of the registers are recorded in the register frames using the slots *tpg_of* and *sar_of*. For example, the "sar_of" slot of the frame for the register P will be filled as shown below:

```
value(reg3,sar_of,outport2).
```

("reg3" and "outport2" are the internal index of the register P and the port OUT of the Multiplier, respectively.)

5.2 Initial State Generation (Phase 2)

If a port does not have available testing resources or constraints are violated, a CUT cannot be made pseudorandom testable. In the example design of Figure 10, two CLBs cannot be made pseudorandom testable. The Barrel Shifter does not have any available resource to act as the SAR. The ALU cannot be made pseudorandom testable

due a the violation of the constraint "A register allocated to an input port cannot be allocated to an output port". Examination of Figure 10 shows that the register ACC is the only testing resource for the port OUT and IN1. In building a test structure frame for this kind of CUT, the value of the test_scheme slot is specified as "incomplete" instead of "pseudorandom". Moreover, the cause of goal failure is specified in the ppg and sar slots. For example, the test structure frame for Barrel Shifter is built as shown below:

```
frame(test_struct_1).
value(test_struct_1,cut,Barrel_sh).
value(test_struct_1,test_scheme,incomplete).
value(test_struct_1,ppg,[[IN,AR0]]).
value(test_struct_1,sar,[[OUT,not_avail]]).
```

In order to make this type of CUT pseudorandom testable, we need to add extra paths and/or extra testing resources. For example, the size of the output of the Barrel Shifter is 32-bits. Therefore, the registers P and ACC can be used as an SAR for Barrel Shifter if an extra path is added from the output of the Barrel Shifter to one of the two registers.

Repairing *incomplete* test structures starts with plan generation. The repair plan is represented by an AND/OR goal tree using the procedural knowledge in the knowledge base. The goal tree for repairing the two incomplete test structures associated with the Barrel Shifter and the ALU is shown in Figure 12. As shown, the necessary operations are defined in each abstraction level and it is very easy to understand the procedure taken.

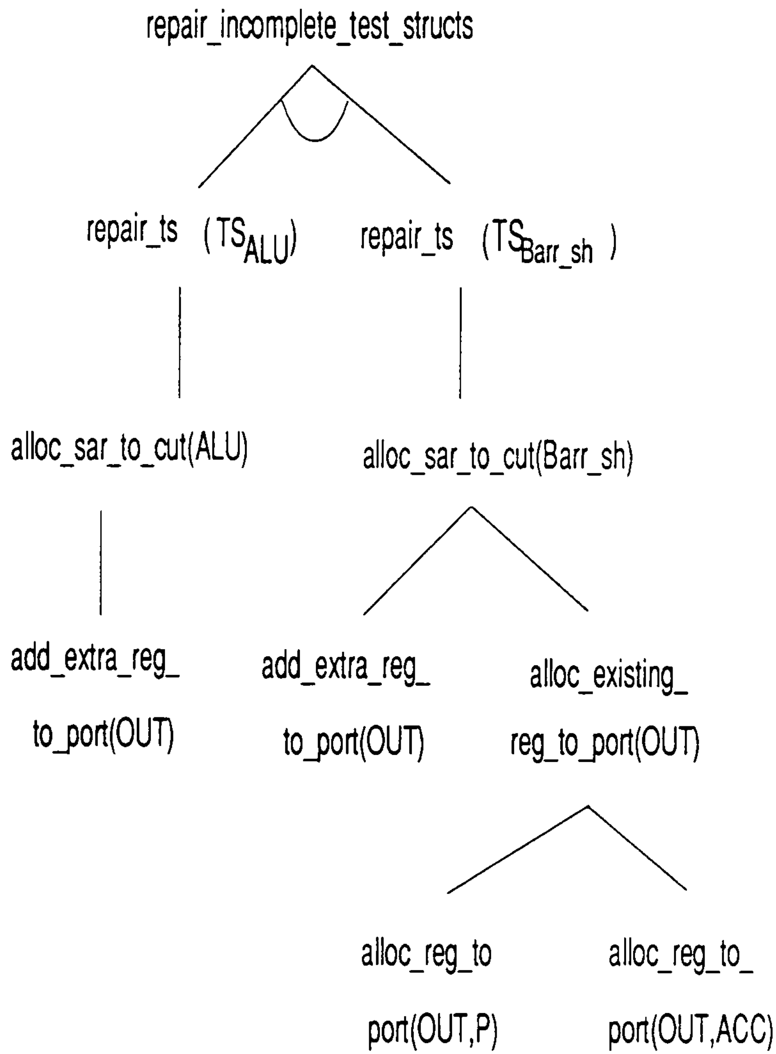


Figure 12. AND/OR Goal tree for repairing incomplete test structures.

As shown, the top level goal "repair_incomplete_test_structs" is expanded into two subgoals: "repair_ts(TS_{ALU})" and "repair_ts(TS_{Barr_sh})". Note that the operator "repair_ts" is defined on test structures. Next, the cause of the incompleteness in the test structure is examined and the subgoal at the circuit level is posted. Since the ALU does not have an SAR, the subgoal "alloc_sar_to_cut(ALU)" is posted. Similarly, "alloca_sar_to_cut(Barr_sh)" is posted as the subgoal to repair the test structure associated with the Barrel Shifter.

The leaf goals in the AND/OR goal tree are defined on the leaf level of the abstraction hierarchy, i.e., the port level. In order to allocate an SAR to the ALU, an extra register must be added because the two 32-bit registers P and ACC are already allocated for testing the ALU. However, these two registers can be used as the SAR for the Barrel Shifter by adding an extra path. Therefore, only one plan, "add_extra_reg_to_port(OUT)" is generated for repairing TS_{ALU}, but there are two options, "alloc_reg_to_port(OUT,P)" and "alloc_reg_to_port(OUT,ACC)" for repairing TS_{Barr_sh}. As shown in the AND/OR goal tree, knowledge for repairing incomplete test structures is represented explicitly and easily with operators in each level of the hierarchy.

The next step is the plan decision, i.e., decide which plan is to be executed. For example, there are two options for repairing the test structure TS_{Barr_sh}, "add extra register" or "allocate existing register". The plan decision depends on the existence of interacting subgoals. If there is no conjunctive subgoal, the decision is made based on the plans for the subgoal itself. However, if there is a conjunctive subgoal, the plan for the other subgoal is also considered. For the example design, in order to accomplish the plan "repair_ts(TS_{ALU})", an extra register must be added. If an extra register must be added, it is desirable to use the extra register as the SAR for Barrel Shifter rather than allocating an existing register. If an existing register (P or ACC) is allocated, an

extra multiplexer must be inserted into the system data path and it will degrade the system performance in terms of speed. This kind of reasoning is performed in the plan decision step. Knowledge about plan decision is represented by production rules. According to the decision made in the plan decision step, the chosen plan is executed. During the plan execution, the test structure is modified. For example, the "sar" slot of the TS_{Barr_sh} is modified as

value(test_struct1,sar,[[out,extra]]).

The generated initial state, i.e., a complete test resource allocation for pseudo-random testing, is shown in Table 2 and Figure 13. As shown, two registers, AR0 and P, are converted into BILBO registers and the other two registers, T and ACC, are converted into PPGs. An extra SAR, EXTRA, and an extra multiplexer is added to the design.

Table 2. Allocation of PPGs and SARs for CLBs.

CLB	PPG	SAR
Multiplier	T ARO	P
Barrel Shifter	ARO	EXTRA
Parallel Shifter	ACC	ARO
ALU	ACC P	EXTRA

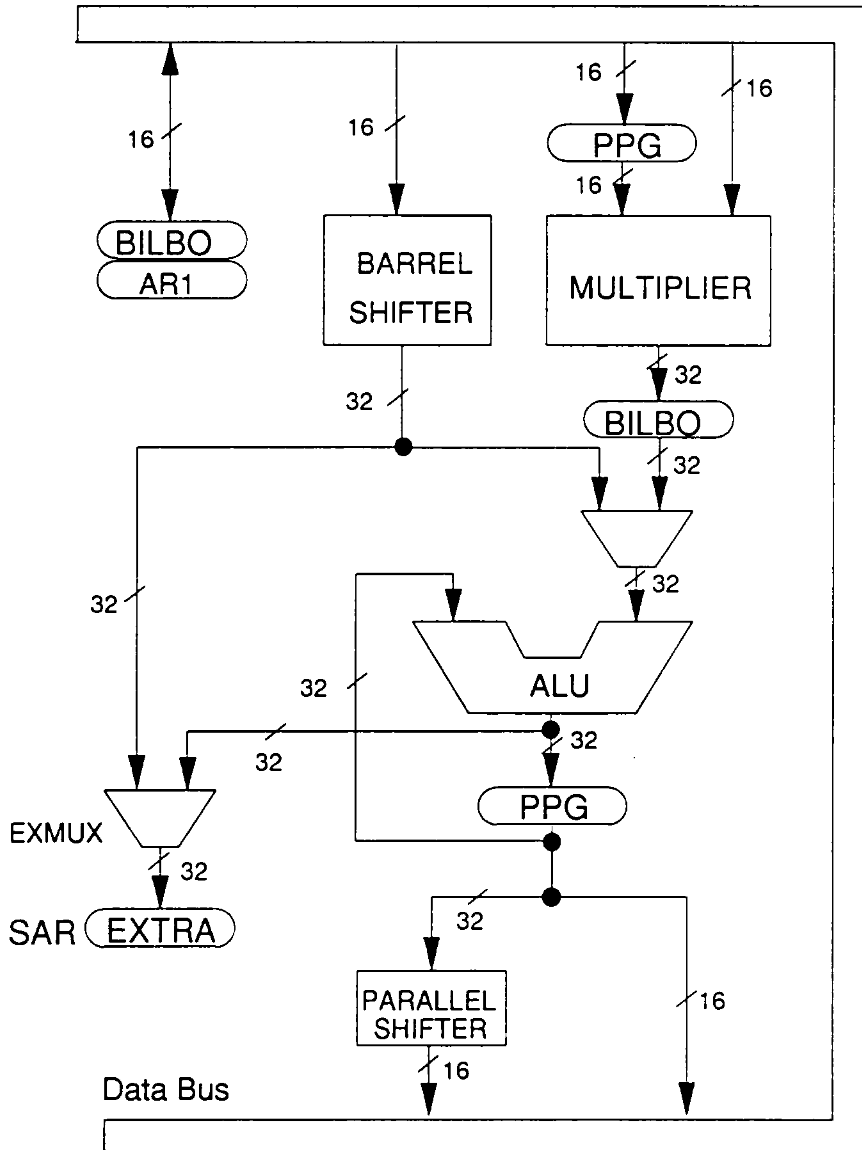


Figure 13. Testing resource allocation for pseudorandom testing.

Chapter 6

Evaluation of BIST Implementation

Whenever, a new state is generated by either test structure building or the transformation of an existing state, the new state must be evaluated to compare it with a goal state. In BIDES, evaluation is performed based on testing time and hardware overhead. Another important factor that a designer may want to know is the additional pin count required by incorporating a BIST scheme. However, it is difficult to estimate the additional pin count using only a register transfer level implementation of BIST since it depends on the low level implementation of the control logic. Thus, the additional pin count is not estimated by BIDES. In this chapter, methods for evaluation of the testing time and hardware overhead are described.

6.1 Evaluation of Testing Time

In BILBO-based BIST, the total testing time depends upon the sharing of testing resources between CLBs. If a testing resource is shared by two CLBs, the two CLBs cannot be tested in parallel. For example, the Parallel Shifter and Barrel Shifter cannot be tested in parallel in the BIST implementation in Figure 13 because of the sharing of AR0 and the Data Bus. Testing of the entire design must then be run in several test sessions because of the test resource sharing. The overall testing time varies with the arrangement of testing sessions. Thus, test scheduling must be performed in order to minimize the testing time for a given BIST implementation. The problem of test scheduling can be stated as follows:

Find a schedule for running tests such that the test for each CLB is run at least once and the total time to run the tests of all CLBs is minimized.

In BIDES, test scheduling is performed using the algorithm proposed in [46]. In the algorithm, a Test Compatibility Graph (TCG) is first constructed based on the sharing of resources. In a TCG, a node represents a CLB. An edge between two nodes indicates that testing of the two CLBs can be performed simultaneously. One thing that should be mentioned in constructing a TCG is that sharing of a PPG should not be considered as a resource sharing conflict since two different CLBs can be tested by the same pseudorandom test patterns. For example, the Barrel Shifter and Multiplier can be tested simultaneously although the PPG AR0 and the Data Bus are used to test both CLBs. The TCG for the BIST implementation in Figure 13 is shown in Figure 14.

Once the TCG is constructed, the test scheduling problem becomes a clique partitioning problem since the minimum number of test sessions corresponds to the minimum number of cliques which cover all the nodes in the graph. Recall that a clique is a maximal complete subgraph of a graph. However, the basic clique partitioning algorithm does not provide an optimal solution since the testing time of each CLB is different; each node in the graph has a weight. For this reason, the minimum number of cliques does not necessarily gives the overall minimum testing time. In order to consider the weight of each node, an algorithm called the *ordered clique partitioning* algorithm is proposed in [46] and used in BIDES. An ordered clique is a clique in which nodes are sorted in the order of their weight. The first step in the algorithm is the generation of all ordered cliques and ordered subcliques. Then, the problem becomes a set covering problem [47]. That is, "Find a set of ordered cliques which covers all nodes in the graph such that the overall testing time is minimized".

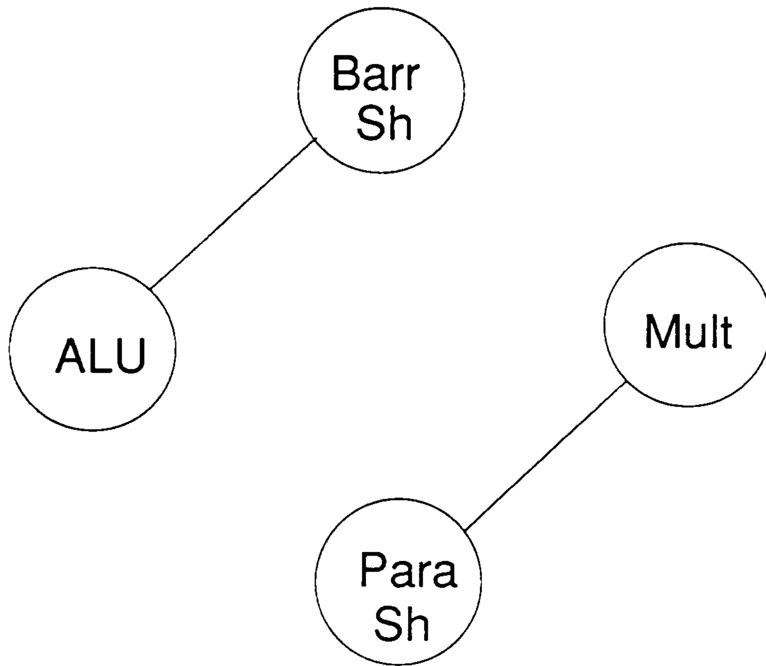


Figure 14. Test compatibility graph for the BIST implementation in Figure 13.

The details of the algorithm can be found in [46].

Table 3 shows the test sessions for the BIST implementation in Figure 13 which were scheduled using the algorithm. As shown, the testing of the four CLBs is performed in two test sessions. In BIDES, the number of pseudorandom patterns, i.e., testing time, for each CLB must be provided by the designer.

6.2 Evaluation of Hardware Overhead

The major sources of hardware overhead in BILBO-based BIST come from the hardware necessary for implementing testing resources, i.e., the PPGs and SARs and the scan-path. Hardware overhead may also come from any extra data paths and extra circuitry such as multiplexers for creating the extra data paths. In BIDES, only the hardware overhead for the testing resources and any extra components is taken into account. The actual silicon area for extra data paths should also be considered, but it is very difficult to estimate at the register transfer level without information about the actual floor plan of the design.

In estimating the hardware overhead for the BIST components, the method proposed in [48] is employed. In the method of [48], various BIST components are implemented using nMOS and the number of extra transistors needed in each BIST component is counted. In BIDES, the number of extra transistors is counted based on the CMOS implementation rather than nMOS and the hardware overhead for each BIST component is represented using a formula [49]. In counting the number of transistors in BIST components, we need to consider two cases of BIST component insertion. If a BIST component is inserted using an existing system register, only the

Table 3. Result of Testing Time Evaluation.

Scheduled Test Session

	Tested CLB	Used Registers				
		AR0	P	ACC	T	EXTRA
Session 1	Barr_sh Multiplier	PPG	SAR	X	PPG	SAR
Session 2	ALU Para_sh	SAR	PPG	PPG	X	SAR

Overall Testing Time $= \max(TT_{Barr_sh}, TT_{Mult}) + \max(TT_{ALU}, TT_{Para_sh})$

$= 2645 + 1857 = 4502$ (patterns)

number of additional transistors should be counted. If a BIST component itself is added to the design, e.g., the SAR EXTRA in Figure 13, then the number of transistors in the entire BIST component must be counted.

As shown earlier, there are three types of BIST components, PPG, SAR, and BILBO, in BILBO-based BIST. These three components are constructed based on the LFSR. Due to the multiple modes of operation of BIST components, it is necessary to have control logic to properly configure the testing resources in each mode. In implementing each component, the control signals for that component are assigned. Then, the control logic and each stage of the component is designed. In the rest of this section, we will show the implementation of each component and derive the formula for estimating hardware overhead. The unit of hardware overhead used is the number of transistors.

6.2.1 Built-In Logic Block Observer (BILBO)

As introduced earlier, BILBO has four functional modes that can be selected by two control signals, B1 and B2 (see Table 4). The first mode is the normal register mode where the BILBO register acts as a normal system register. Next, the BILBO register can be operated as a shift register. In this mode, the BILBO can be used as part of a scan-path; shifting data from the input, scan-in, to the output, scan-out. The last two modes are based on the LFSR. Depending on the control signals, the BILBO can be run as a PPG or as an SAR. The circuit diagram of the BILBO register is shown in Figure 15. As shown, there are two signals fed to each stage (bit position), normal DATA IN and the shifted-in signal, i.e., the signal from the previous stage. In the PPG and Scan mode, the shifted-in signal is selected. In the SAR mode, the two signals are EXORed and fed to the D flip-flop. The control logic for selecting signals is included

Table 4. Control Signals for BILBO.

B1	B2	Modes
0	0	PPG
0	1	Scan
1	0	Load
1	1	SAR

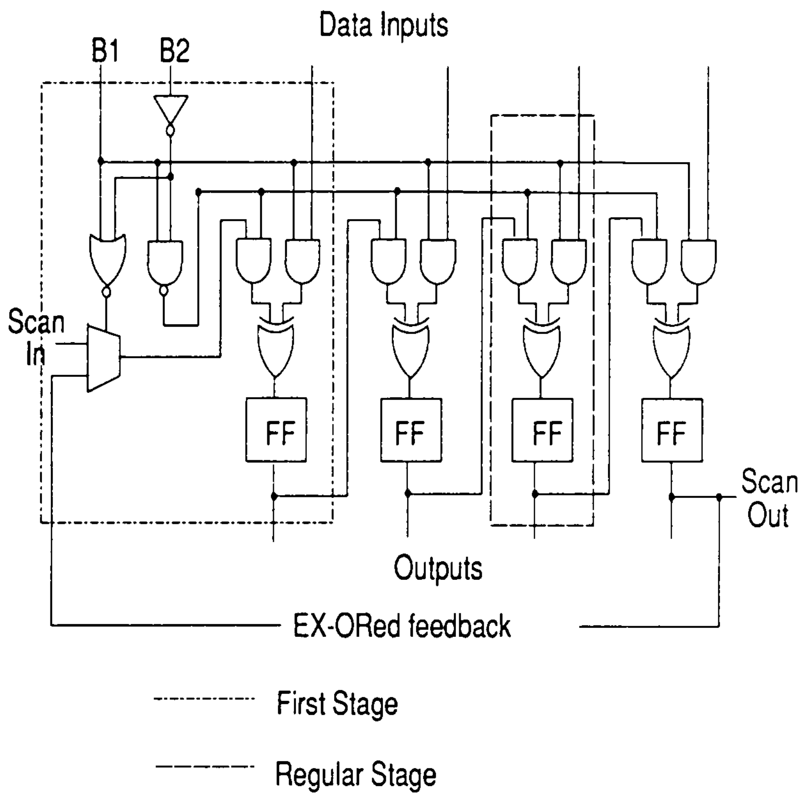


Figure 15. BILBO.

in the first stage. The remaining stages are the same as the first stage except for the control logic. Therefore, the BILBO can be partitioned into three parts, the first stage, regular stage, and the EXORed feedback.

In order to implement the BILBO register by modifying the system register, all circuitry except for the D flip-flops must be added. In Table 5, the additional gates for the first stage are shown. The number of transistors is counted based on the CMOS implementation. The CMOS implementation of each gate can be found in [50]. In order to implement the regular stage, we need to add two AND gates and one EXOR gate. Therefore, we need total 18 extra transistors per regular stage. The number of EXOR gates in the feedback path depends upon the primitive polynomial used to implement the BILBO. Most of the primitive polynomials have three feedback terms [12]. Therefore, we assume the use of three EXOR gates (24 transistors) in the feedback logic. Then, the total number of additional transistors for an n-bit BILBO will be

$$\text{Overhead (Modified BILBO)} = (n-1) * 18 + 34 + 24.$$

A BILBO register that is added to a design only for the purpose of testing is called an extra BILBO register. In the situation when an extra BILBO register must be added, the transistor count for the D flip-flop for each stage including the first stage should be added to the overhead count. The use of a 2-phase latch with an eight-transistor implementation is assumed. The control logic is designed by simply excluding the normal operation mode from the modified BILBO register since the extra BILBO register does not need to operate normal mode. The first stage of the extra BILBO register is shown in Figure 16. From the diagram, the overhead for the n-bit extra BILBO can be derived as

$$\text{Overhead (Extra BILBO)} = (n-1) * 26 + 38 + 24.$$

Table 5. Overhead for the First Stage of the Modified BILBO.

Additional Gates	No. Of TRs
2 AND	10
1 NAND	4
1 NOR	4
1 Inverter	2
1 EXOR	8
1 MUX	6

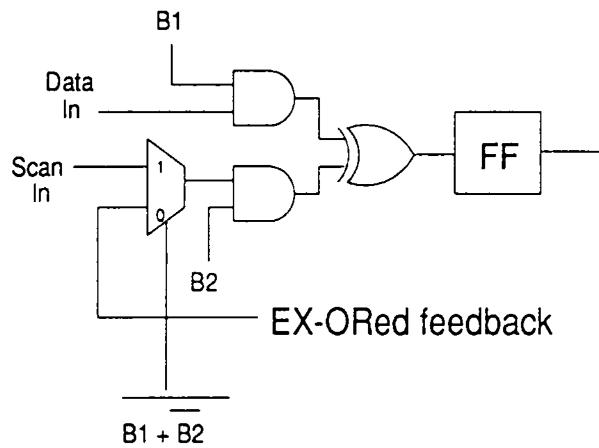


Figure 16. The first stage of extra BILBO.

6.2.2 Signature Analysis Register

When a register is to be used as an SAR only, not BILBO, the PPG mode of the BILBO can be excluded. However, two signals, DATA IN and the shifted-in signal need to be EXORed together since DATA IN signal is an output of a circuit being tested. Thus, the regular stage of an SAR is identical with the regular stage of the BILBO. The first stage is changed since the PPG mode is not necessary. Since the collected signature needs to be scanned-out after testing, the Scan mode is still necessary. The circuit diagram of the first stage is shown in Figure 17. The overhead for the n-bit SAR can be estimated as

$$\text{Overhead (Modified SAR)} = (n-1) * 18 + 24 + 24.$$

If an SAR is used only for testing purpose as in the case of an extra BILBO, only two operational modes, the SAR and Scan mode are necessary. Thus, a single control signal is enough. The first stage of the extra SAR is shown in Figure 18. The regular stage of the extra SAR is implemented by excluding the multiplexer used for distinguishing the EXORed feedback signal and the Scan-In data. Note that the D flip-flop should be counted as part of the overhead for the extra component. The overhead is then given by

$$\text{Overhead(Extra SAR)} = (n-1) * 22 + 28 +24.$$

6.2.3 Pseudorandom Pattern Generator

As mentioned earlier, PPG is just another name for an LFSR. The difference between an LFSR and a normal shift register is that the LFSR has EXORed feedback which is shifted into the first flip-flop. If a system register is to be converted into a PPG, it is only necessary to isolate the normal DATA IN signal and the shifted-in

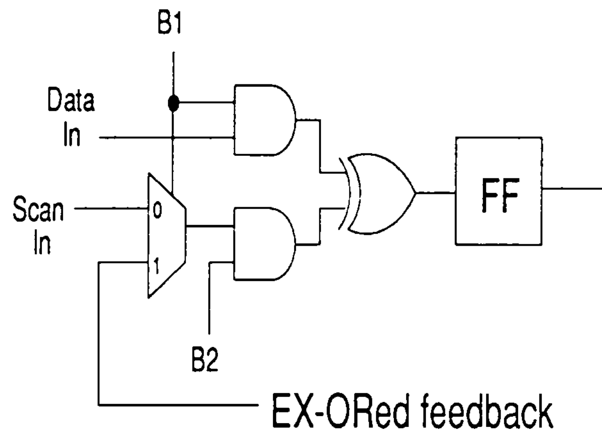


Figure 17. The first stage of modified SAR.

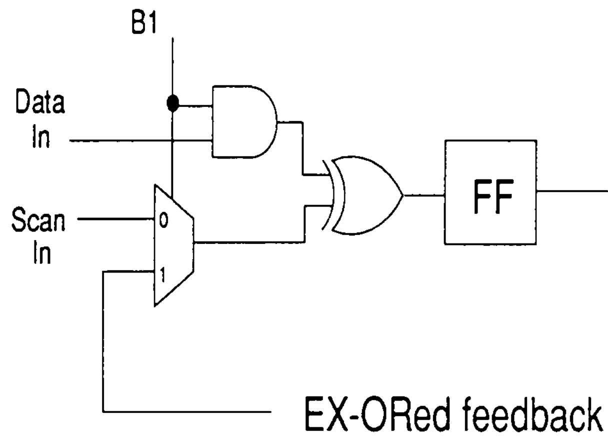


Figure 18. The first stage of extra SAR.

signal. Therefore, the regular stage can be implemented by simply inserting a multiplexer. It is again assumed that the PPG contains the Scan mode for initialization. The initial pattern for the PPG can be scanned-in using the scan-path. Then, the first stage needs only another multiplexer as compared to the regular stage shown in Figure 19 for separating the scan-in signal the EXORed feedback. Hardware overhead for the n-bit modified PPG is shown below:

$$\text{Overhead (Modified PPG)} = (n-1) * 4 + 16 + 24.$$

The first stage of the extra PPG with the Scan-mode is shown in Figure 20. The regular stage of the extra PPG is just a D flip-flop. Therefore, the overhead is

$$\text{Overhead (Extra PPG)} = (n-1) * 8 + 14 + 24.$$

6.2.4 Scan Flip-flop

If an n-bit system register is to be transformed into a scan register, it is necessary to include a 2:1 multiplexer to isolate DATA IN from the scan signal. Therefore, the overhead for the n-bit scan register is 4n transistors (a 1-bit 2:1 MUX needs 4 transistors).

6.2.5 Total Hardware Overhead

Using the formulae derived above, the total hardware overhead is estimated by summing the hardware overhead for the BIST components and the extra components added to the design. The hardware overhead for each BIST component and the overall hardware overhead for the initial state in Figure 13 is shown in Table 6. Note that the register AR1 is not used as a testing resource. It is simply converted into a scan flip-flop.

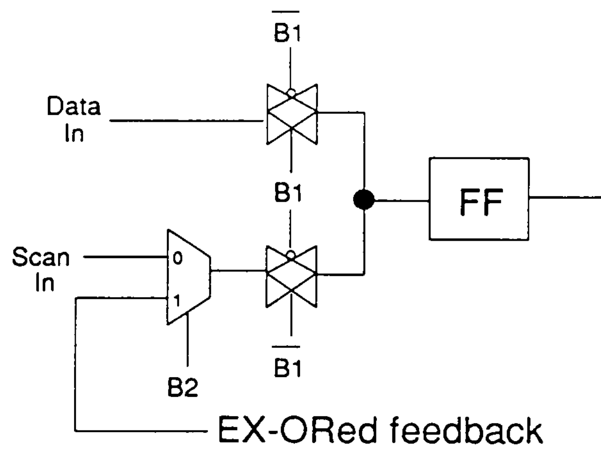


Figure 19. The first stage of modified PPG.

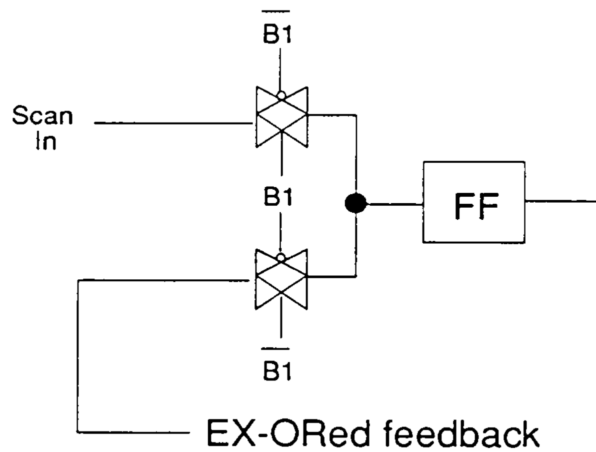


Figure 20. The first stage of extra PPG.

Table 4. Result of Hardware Overhead Evaluation.

Component	Function	Hardware Overhead (TRs)
ACC	PPG(32)	164
P	BILBO(32)	616
AR0	BILBO(16)	328
T	PPG(16)	100
AR1	Scan FF(16)	64
EXTRA	SAR(32)	734
EXMUX	MUX(32)	128
Total Hardware Overhead		2134

Chapter 7

Regeneration of BIST Structure

If the current BIST implementation cannot meet the design objectives, an alternative is sought based on the difference between the current state and a goal state. Since there are two factors in the difference, hardware overhead and testing time, the search needs to be directed toward one of the following three directions.

1. Both testing time and hardware overhead need to be reduced.
2. Only testing time needs to be reduced.
3. Only hardware overhead needs to be reduced.

Among the above direction options, only two directions, reducing testing time or reducing hardware overhead, are considered by BIDES. This is because it is almost impossible to reduce both testing time and hardware overhead simultaneously. As shown in the example in Chapter 2, there exists a trade-off between testing time and hardware overhead. Based on the result of an evaluation of the current state, the search direction, either "reduce testing time" or "reduce hardware overhead", is decided upon by the designer. Then, a child of the current state is generated. The new state will be evaluated and the result will be provided to the designer for approval or the initiation of further modification.

7.1 Reducing Hardware Overhead

Hardware overhead is reduced by reducing the number of testing resources. There are two kinds of elimination that can take place: removal of a register which is added for testing purpose only, e.g., EXTRA in Figure 13, and restoration of a normal register, which was previously transformed into a testing resource.

Testing resource elimination starts with the selection of a register to be eliminated. The designer can select a candidate register to eliminate. Otherwise, a register is selected by BIDES using pre-defined rules. The prime candidate for elimination is an extra register since it generates the most significant portion of the total hardware overhead. For the existing registers, the selection is made based on the difficulty of elimination. Difficulty of elimination is measured by the number of test structures in which the register is used. In the example design, the extra register EXTRA is selected first. Next, the register T will be selected because the register T is used only for testing the Parallel Shifter.

Register elimination is also performed using hierarchical planning. The plan generated for removing EXTRA is shown in Figure 21. As shown, the plan generation starts with the posting of goals on the test structure level using the operator "modify_ts_without_reg". This operator has two arguments, the register to be eliminated and the test structure to be modified. In order to eliminate EXTRA, two test structures, TS_{Barr_sh} and TS_{ALU}, must be modified. Then, the role of the register in the test structure is identified and the goal is specified on the CUT level. For example, since EXTRA is the SAR for the ALU, the goal "change_sar_of_cut(ALU)" is posted in order to achieve the upper level goal "modify_ts_without_reg(TS_{ALU}, EXTRA)". The only possible option to change the SAR of the ALU is to use a feedback testing scheme to test the ALU since there is no other resource which can be used as the SAR of the ALU. The use of feedback testing is implemented by allocating the register ACC to the port OUT of the ALU. This is specified with the operator "alloc_reg_to_port(OUT,ACC)". In order to make Barrel Shifter self-testable without EXTRA, we have two options: the use of P or ACC as the SAR. However, both options require the addition of a multiplexer in the system data path.

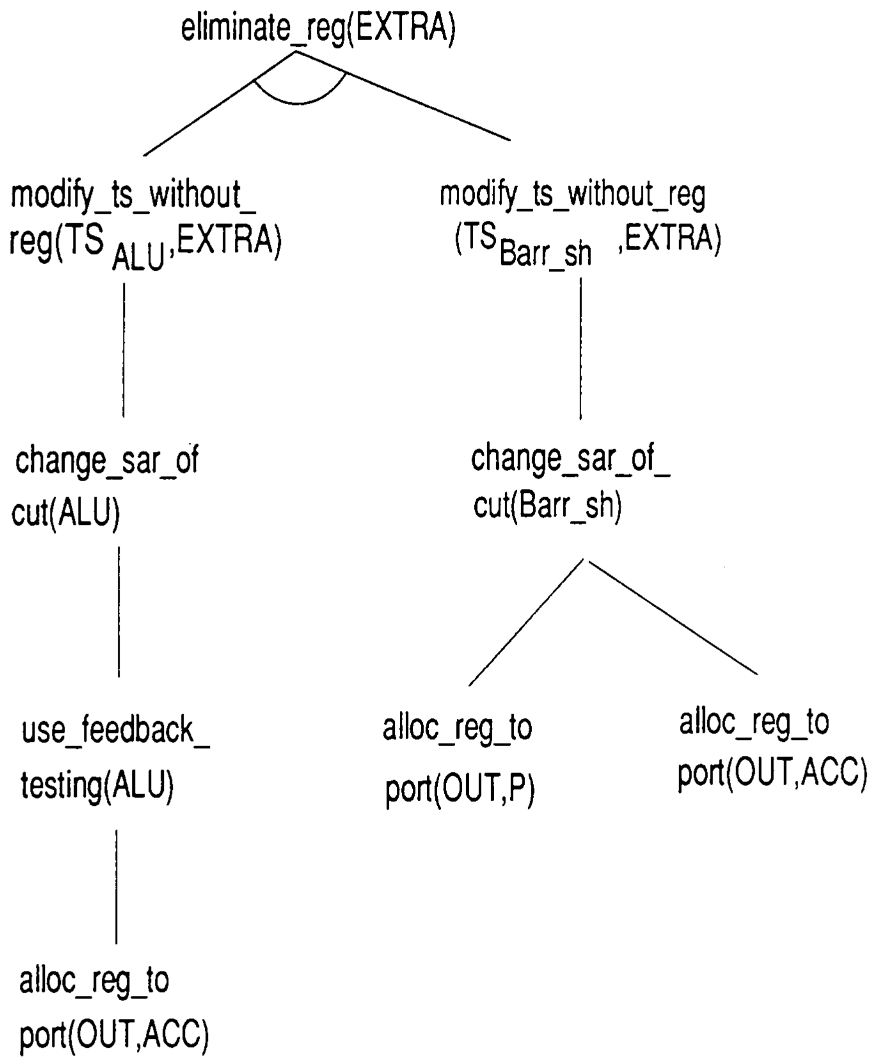


Figure 21. Plan for eliminating EXTRA.

As shown, the plans are generated through the use of abstraction hierarchy. Therefore, another option for a goal, e.g, eliminate_reg, can be easily added to the goal tree by defining an operator in each abstraction level. There is no need to change the control mechanism of BIDES. This is a strong advantage of using hierarchical planning based on abstraction hierarchy.

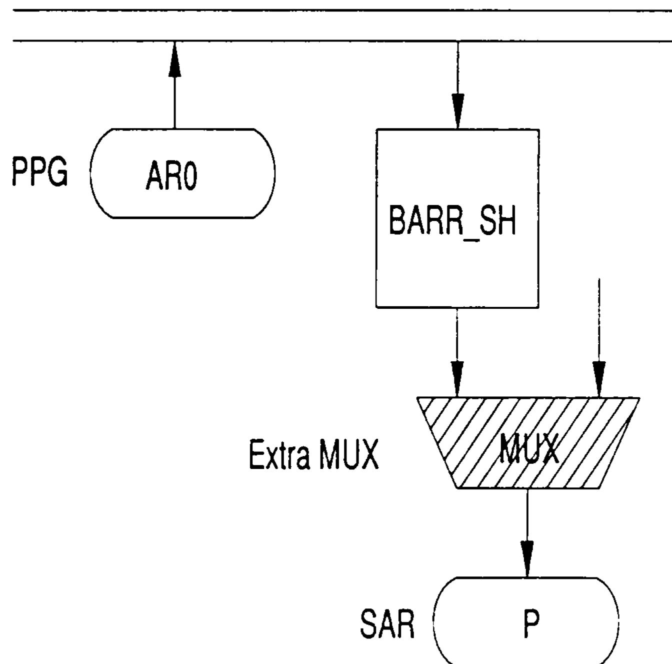
As mentioned earlier, a plan to be executed is chosen in the plan analysis step. For modifying TS_{ALU} without EXTRA, the use of a feedback testing scheme is the only possible option. Therefore, this option is chosen. However, there are two options for changing the SAR of the Barrel Shifter. Namely, allocate P or ACC. In this case, the selection is made based on two factors, the type of the candidate registers and the difficulty of the elimination of newly allocated register. In this specific example, we need an SAR for the barrel shifter. If a candidate register is already used as an SAR for another CLB, then no extra circuitry is needed except for an extra path. If a candidate register is used as only a PPG, then we need extra circuitry for converting it into a BILBO. In other words, the first heuristic used for selection is to select a register which has the same type. In this way, we can reduce the hardware overhead as much as is possible. However, for the SAR of the Barrel Shifter, P and ACC are used as an SAR. Therefore, selection cannot be made using the first heuristic.

The second heuristic is based on the difficulty of eliminating a newly allocated register. The designer may want to further reduce hardware overhead after the elimination of the register being considered, i.e., another register may need to be eliminated. The next candidate register may be the register which was just used to reduce hardware overhead, e.g., P or ACC may need to be eliminated later. Therefore, when considering the further elimination of registers, it is better to choose a register that is a part of the

fewest number of test structures. For example, P is used in two test structures, TS_{MULT} and TS_{ALU}. However, for this example, ACC is also used in two test structures, TS_{ALU} and TS_{Para_sh}. In this case, register P is chosen arbitrarily.

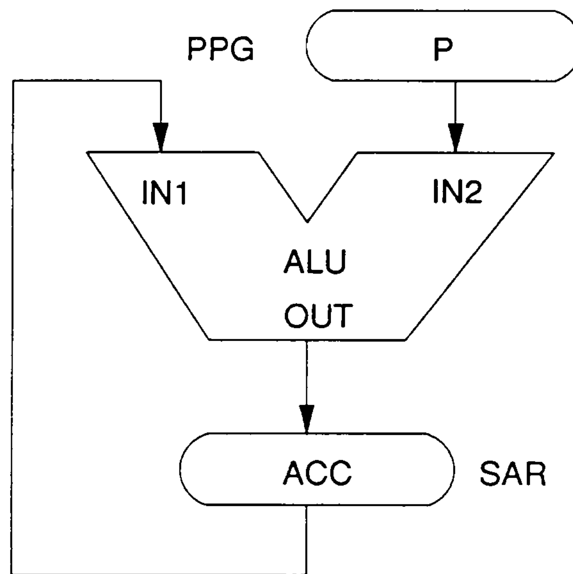
The modified design after the elimination of register EXTRA is shown in Figure 22. As shown, the register EXTRA and the extra multiplexer EXMUX are eliminated. By eliminating these two components, hardware overhead is reduced by 15%. However, another extra multiplexer is inserted into the system data path. This will degrade the system performance. Moreover, the modified TS_{ALU} needs a fault simulation to decide on the test length necessary to obtain a satisfactory fault coverage of the ALU based on the feedback testing scheme. The number of test patterns in the feedback testing scheme is larger than that of pseudorandom testing. This longer testing time is indicated by the symbol α, which is greater than 1.0, in Figure 22. This kind of information is returned to the designer with the result of test session arrangement.

If the designer needs to further reduce the hardware overhead, then BIDES tries to eliminate another register. If a BIST implementation contains no extra register, a register is selected based on the difficulty of elimination. For example, the register T will be selected in the example design since T is used only as the PPG of the Multiplier. Other registers in the design are involved in at least two test structures. After selecting a register, the same steps plan generation, plan analysis, and plan execution, are taken to eliminate the register. The plan generated for eliminating the register T is shown in Figure 23. As shown, there in only one option, allocate AR0 to the port Mult1 of Multiplier. The register ACC seems to be a candidate to substitute for T. However, ACC cannot be used since test patterns would have to be applied from ACC through the Data Bus. Notice that the Data Bus is also used by AR0 for providing test



(a) Modified TS_{Barr_sh}

Figure 22. Modified test structures after eliminating EXTRA and results of evaluation.



(b) Modified TS_{ALU}

Total Hardware Overhead = 1852 Transistors.

Total Testing Time $= TT_{Barr_sh} + TT_{Mult} + TT_{Para_sh} + \alpha * TT_{ALU}$

$= 332 + 2645 + 209 + \alpha * 1857$

$= 3186 + \alpha * 1857$

$(\alpha > 1.0)$

Figure 22. Modified test structures after eliminating EXTRA and results of evaluation (con'd).

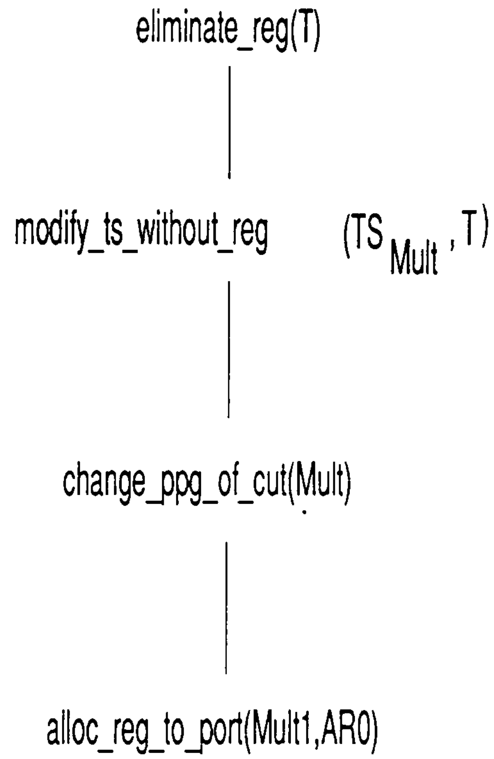


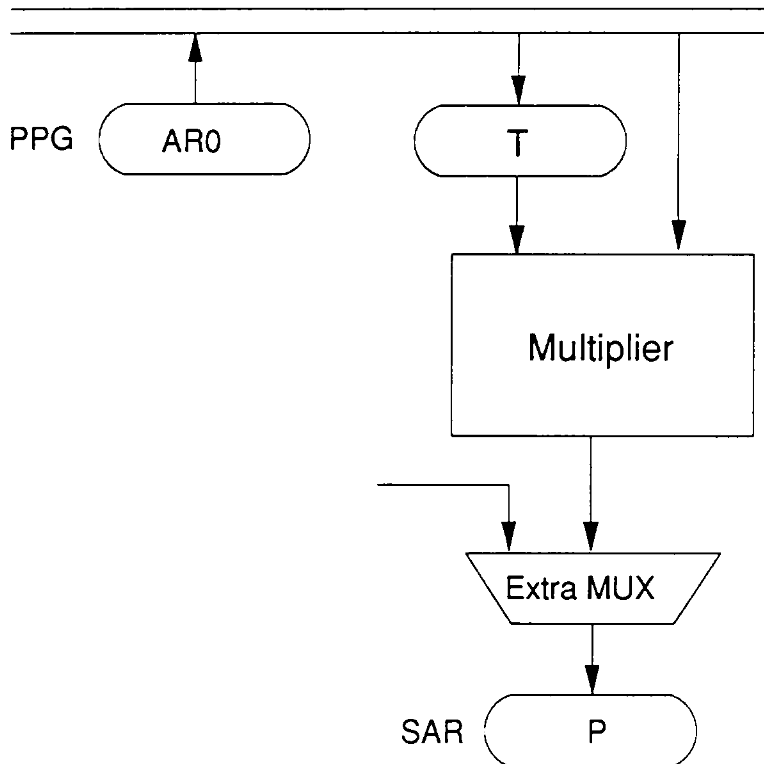
Figure 23. Plan for eliminating T.

patterns to the port Mult2. If AR0 is used as the PPG for the port Mult2, the same register is used as the PPG for the two different ports, Mult1 and Mult2, of a single CLB. Obviously, this is in violation of the constraints expressed earlier on the pseudorandom testing. However, AR0 is the only register that can substitute for T. Therefore, BIDES will allocate AR0, but the problem with using AR0 is then related to the designer. A fault simulation is required to identify the necessary test length since the same patterns will be applied to the two different ports of the Multiplier. The result of this elimination is shown in Figure 24.

Register elimination can be continued until no further register can be eliminated. The example design needs at least three registers for self-testing. Therefore, hardware overhead cannot be reduced anymore than what is shown in Figure 24. This information is informed to the designer if the designer wants further reduction in hardware overhead.

7.2 Reducing Testing Time

In attempting to reduce testing time, one restriction is put on the designer by BIDES. Since hardware overhead is still a major concern for the designer when incorporating BIST, it is unreasonable to expect a designer to reduce testing time without regard to increased hardware overhead. For this reason, BIDES only considers the use of a cascade testing scheme for reducing testing time. As introduced in Chapter 2, the cascade testing scheme can reduce testing time when two modules are cascaded together through a register. The cascade testing scheme requires that the register between the two modules under test be an SAR. Previously, this register had been



Total Hardware Overhead = 1816 Transistors.

Total Testing Time $= TT_{Barr_sh} + \beta * TT_{Mult} + TT_{Para_sh} + \alpha * TT_{ALU}$

$= 332 + \beta * 2645 + 209 + \alpha * 1857$

$(\alpha, \beta > 1.0)$

Figure 24. Modified TS_{Mult} after eliminating T and results of evaluation.

transformed into a BILBO in the test structure building step. Thus, the use of cascade testing to reduce testing time does not require any additional hardware overhead. Rather the hardware overhead is reduced slightly.

Let us consider the reduction of testing time from the initial state in Figure 13 by using cascade testing scheme. In the example design in Figure 10, the Multiplier is driving the ALU through the register P. If P is just transformed into just an SAR rather than a BILBO, the test patterns applied to the port IN2 of the ALU are the contents of the SAR, not the pseudorandom patterns. Since the patterns generated by an SAR can be repeated as mentioned in Chapter 2, the individual testing time of the ALU is longer than that of pseudorandom testing. However, the ALU is tested in parallel with the Multiplier, the overall testing time can be reduced.

Cascade testing is implemented by calling the procedure "combine_tss_for_cascade_testing" for the two cascaded test structures as shown below:

```
combine_tss_for_cascade_testing(TS1,TS2) :-
    cascade(TS1,TS2,
            TS_Driving,TS_Driven,Intermidiate_Reg),
    build_cascaded_ts(TS_Driving,TS_Driven,Intermidiate_Reg),
    modify_ts(cascade_testing,driving,TS_Driving,Intermidiate_Reg),
    modify_ts(cascade_testing,driven,TS_Driven,Intermidiate_Reg).
```

The above procedure first checks that the two test structures are actually cascaded. If so, the driving test structure, the driven test structure, and the intermediate register are identified. This is done by the procedure "cascade". Then, the procedure "build_cascaded_ts" produces a <u>macro test structure</u> which consists of the two micro

test structures. Next, the individual test structures are modified so as to perform cascade testing. By defining the macro test structure, the knowledge about cascade testing can be represented explicitly. The frame representation of the macro test structure produced is shown below:

frame(test_struct5).

value(test_struct5,is_a,macro_test_struct).

value(test_struct5,test_scheme,cascade_testing).

value(test_struct5,driving_ts,TS_{Mult}).

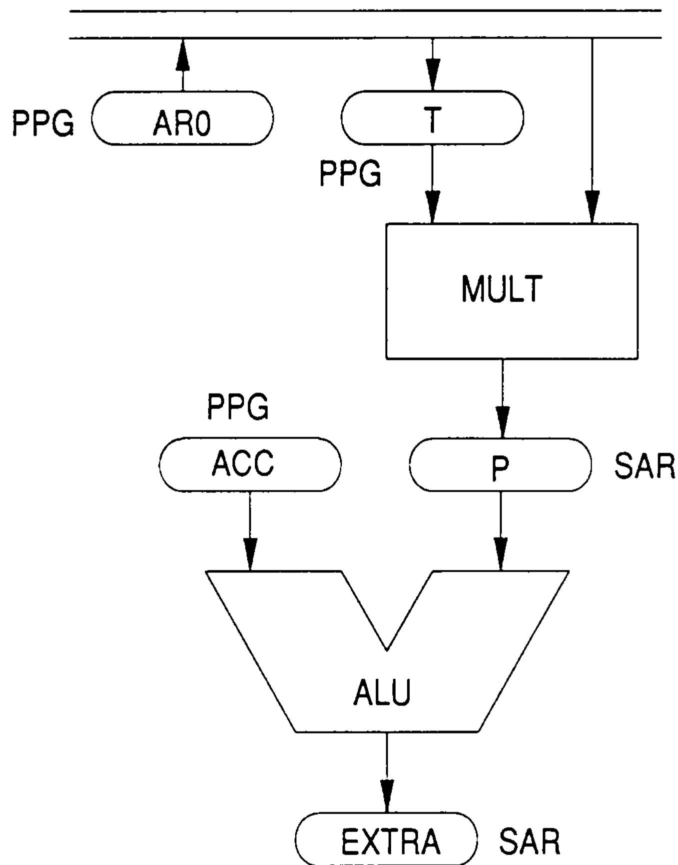
value(test_struct5,driven_ts,TS_{ALU}).

value(test_struct5,intermidiate_register,P).

The testing time for two cascaded CLBs when using the cascade testing scheme is max(TT_{Mult}, α * TT_{ALU}) since two CLBs are tested in parallel. The parameter α which is greater than 1.0 indicates that the individual testing time of the ALU is larger than that of pseudorandom testing (TT_{ALU}) since the test patterns of ALU are generated by the SAR. Therefore, the value of α must be estimated in order to estimate the overall testing time. In BIDES, the test length for the ALU, i.e., the value of α, in the cascade testing is estimated using a formula which estimates the number of different patterns generated by SARs [6] as shown below:

$$\text{Test Length of ALU} = \frac{\log\left(1 - \frac{TT_{ALU}}{2^n}\right)}{\log\left(1 - \frac{1}{2^n}\right)}$$

The results of modifications performed in order to reduce testing time are shown in Figure 25. As shown, the use of cascade testing made it possible to test the ALU with the Multiplier in parallel. However, this causes separate testing of the Barrel Shifter and the Multiplier since the Barrel Shifter cannot be tested in parallel with the



Total Hardware Overhead = 2124 Transistors.

Testing Time $= TT_{Barr_sh} + TT_{Para_sh} + max(TT_{Mult}, 1.012 \times TT_{ALU})$

$= 541 + max(2645,1879)$

$= 3186$

Figure 25. Parallel testing of the Multiplier and the ALU using the SAR (P) as the test pattern generator for ALU.

ALU because of sharing of register EXTRA. Thus, it requires three test sessions to test all four CLBs. However, in spite of the increased number test sessions, the overall testing time is reduced significantly because parallel testing of the Barrel Shifter and the ALU is now possible. Moreover, testing time is reduced without any additional hardware overhead. Further reduction of testing time is not possible since there is no other portion of the circuit where cascade testing can be incorporated.

7.3 Scan-Path Organization

After finding a satisfactory test resource allocation, scan-path organization is performed. As mentioned earlier, the scan-path is used to test registers and to observe the collected signatures. In scan-path organization, an attempt is made to set up a scan-path so that all of the registers in a design are configured as a single shift register chain. Under this assumption, the problem of scan-path organization is to find an optimal order of registers so that the wire length of a scan-path can be minimized. This is the well-known shortest path problem on directed graphs. On a graph, a node corresponds to a registers and the cost of an edge corresponds to a geometrical adjacency between registers.

We formulated the problem as the all-pairs shortest paths problem [49] and Floyd's algorithm [49] is used for solving the problem. For each pair of registers, the shortest path which goes through all of the registers is found and the length of the path is calculated. Then, the path with the shortest length is chosen. Information on registers adjacency is obtained either from user specification or from a back-annotation of the IC layout.

For the example circuit, the information about the geometrical adjacency between registers is not available. Thus, we arbitrarily assign values for geometrical adjacency between registers and the algorithm is applied. The result of the scan-path organization for the BIST implementation in Figure 13 is shown below:

Scan-In --> AR1 --> AR0 --> T --> P --> EXTRA --> ACC --> Scan-Out

7.4 Control Signal Distribution

In each test session, a register operates in one of the four functional modes, PPG, SAR, Scan, and the Parallel Load. Since two bits are enough to control these four functional modes, 2n control lines are sufficient to control n registers. The number of control lines can be reduced by finding registers which can be controlled by the same control signals. In BIDES, this reduction of the number of control lines is performed using the procedure proposed by Kalinowki *et al.* [51].

As described in [51], n+1 control lines are needed to control n BILBOs assuming that Scan and Parallel Load modes are not operated simultaneously. One line, say C_1, which is distributed to all BILBOs, is used to distinguish between Parallel Load/Scan and PPG/SAR and n additional lines, say, C^i_2, i=1,2, ..., n, are used to distinguish between Parallel Load and Scan, or PPG and SAR for each BILBO. Under this assumption, the procedure of control signal distribution is primarily concerned with the reduction of the number of C_2 control lines. The procedure is illustrated using the test session arrangement shown in Table 3 .

The main objective of the procedure is simply to find groups of BILBOs, called *control partitions*, which can be controlled by the same C_2 control lines. In finding a control partition, additional local control logic for each BILBO is assumed. Two

BILBOs, AR0 and P, can be controlled by the same C_2 line, even though the operational mode in each test session is different. It is called *comparable* if BILBOs can be controlled by the same C_2 line such as AR0 and P. An example of two BILBOs that are not comparable is shown in Table 7. These two BILBOs cannot be controlled by the same signal even with local control logic since the value of C_2 should be the same in session 2 and session 3, but should be different in session 1.

One thing that should be mentioned about control signal distribution is that only BILBOs, i.e., registers that are used as both PPG and SAR, need to be considered in finding control partitions. There are three types of registers in BIST other than BILBOs. The first type is a register which is used as a PPG or an SAR, but not both. For example, the register ACC is used only as a PPG in the test resource allocation in Figure 13. For this type of register, we do not need to distinguish between modes PPG and SAR. The mode is fixed in the hardware. Therefore, the C_2 signal of any BILBO can be used to control this type of register. The second type of register is one which is used only in the testing mode. An example is the register EXTRA. Since this type of register has just two operational modes, PPG/SAR and Scan, the register can be controlled by only the C_1 signal. The last type of register is one such as AR1 which is not used in testing CLBs at all. Since we are assuming the scanning of all operational registers in a design, there are two functional modes, Scan and Parallel Load, in this type of register. This type of register can be controlled by the C_2 signal of any BILBO with local control logic.

In the BIST implementation in Figure 13, there are two BILBOs, AR0 and P, and they are comparable. Therefore, we need only two control signals to control all registers.

Table 7. Two Incomparable BILBOs.

	BILBO 1	BILBO 2
Session 1	PPG	SAR
Session 2	PPG	PPG
Session 3	SAR	SAR

Chapter 8

Study on Cascade Testing

As introduced in Chapter 2 and Chapter 7, the cascade testing scheme can be used to reduce testing time as compared to the pseudorandom testing when two CLBs are casecaded together. The basic idea behind cascade testing is the use of an SAR as a PPG. However, in order to use an SAR as a PPG, it must be certain that the characteristics of the patterns generated by the SAR should be similar to pseudorandom patterns. In this chapter, we investigate the characteristics of patterns generated by an SAR. First, the characteristics are studied analytically. Then, the results of the analysis are reviwed using experiments with real circuits.

8.1 Background

In digital systems, it is common that one CLB drives another CLB through a register as shown in Figure 2 in Chapter 2. Conventional pseudorandom testing can be implemented by transforming the three registers, R1, R2, and R3, into a PPG, a BILBO, and an SAR respectively, as shown in Figure 2 in Chapter 2. Clearly, the two CLBs A and B cannot be tested in parallel using the BILBO approach because of the conflict in the roles imposed on register R2. Two sessions are required to test the two CLBs. A method of testing the two CLBs in parallel was proposed in [33]. In this method, the two CLBs are cascaded directly through R2 as shown in Figure 26. R1 and R3 are modified to act as a PPG and an SAR, respectively. R2 is not modified and continues to operate as a normal register. A single signature captured in the

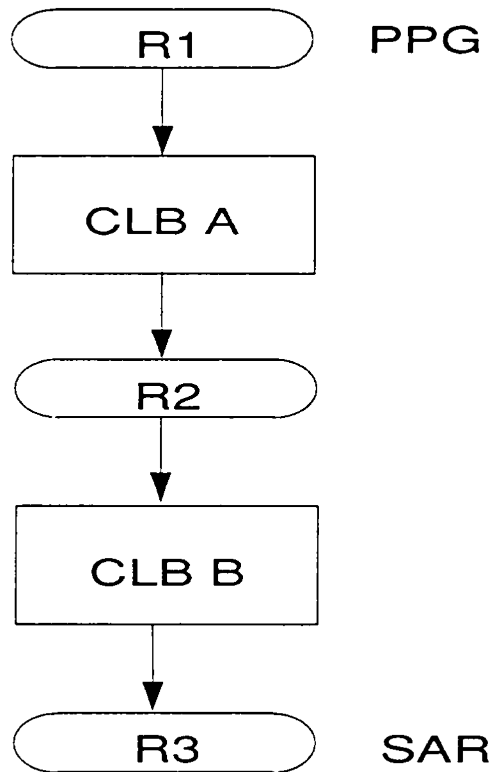


Figure 26. Parallel testing of two cascaded CLBs using a single SAR.

modified R3 is observed in order to evaluate the overall test of both CLBs. Our experiments (the results are shown in Section 8.4) indicate that this method suffers low fault coverage due to the poor observability of CLB A. Moreover, CLB A may not generate all of the necessary test patterns needed to test CLB B.

Another approach for testing the two CLBs is to use the signature of CLB A as the test pattern generator of CLB B as introduced in Figure 3 of Chapter 2. Registers R2 and R3 need to be modified to act only as SARs for CLB A and CLB B, respectively. During testing, the PPG modified from R1 applies pseudorandom patterns to CLB A and the test responses of CLB A are collected by the SAR R2. At the same time, the contents of R2 (we call them patterns) serve as test patterns for CLB B. Signatures of CLB A and CLB B are observed to evaluate the test. The key assumption employed in this approach is that the patterns generated by the SAR during testing are close to random patterns.

The research on the use of signature registers as test pattern generators has been reported in the literature. Carter reported that the use of SARs as PPGs is an open problem [52]. Bardell and McAnney reported that SARs were successfully used as test pattern generators in a multi-chip environment [20]. Krasniewski and Pilarski proposed a BIST architecture in which a circular shift register chain (not an SAR) generates test patterns and at the same time collects test responses [53]. Through various experiments, they showed that the architecture is efficient in testing time and hardware overhead. Beucler and Manner proposed a method in which the SAR of a CLB is used as a test pattern generator for the CLB itself [7] as in the feedback testing scheme. It was claimed that they could achieve reasonably high fault coverage for three combinational circuits used for the experiments. A similar method to Beucler and Manner's method was proposed by Gannet to test a multiplier [8]. In this method, a circuit,

called a "feedback processor", was used to collect test responses and to generate test patterns. Wang and McCluskey showed that SARs can be used as PPGs for BIST design of sequential circuits [54].

Although various work was done on using the SAR as test pattern generators, there has been no intensive work reported on the analysis of the approach. The question with regard to the use of SARs as PPGs is "How good is an SAR as a test pattern generator?". Two aspects, the randomness of the patterns and the repetition of the same patterns, are the most critical factors in determining the quality of an SAR as a test pattern generator. We study these two aspects through analyses in the next section. In the analyses performed, we do not consider the use of an SAR in the feedback testing scheme shown in Figure 4. We only consider the case where the SAR of a CLB is used as a PPG of another CLB as shown in Figure 2 in Chapter 2.

8.2 Characteristics of Patterns Generated by SAR

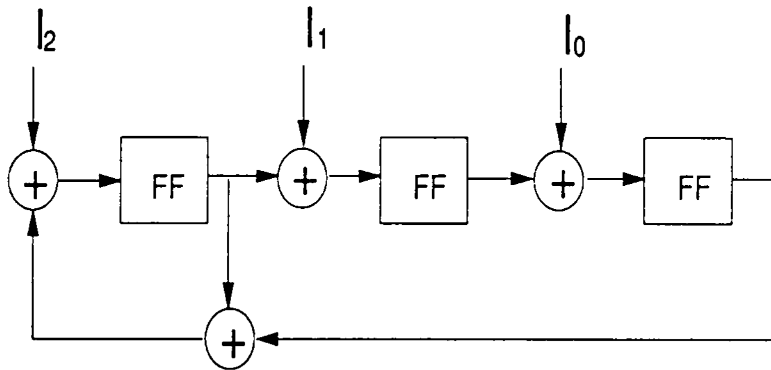
In this section, we show that the patterns generated by an SAR (which normally collects and compacts the test responses of a module) are random numbers selected from the entire range with replacement. As long as the patterns generated by the SAR do not repeat too frequently, they could be used as pseudorandom test patterns. We will study this possibility later in this section. The next pattern generated by an SAR, i.e., the next state of the SAR, is determined by the current state and the input (called the input pattern) applied to the SAR. To simplify the analysis, we assume that the probability of applying an particular input pattern to an SAR does not depend upon the state of the SAR.

Figure 28 shows the state transition diagram of a 3-bit SAR in Figure 27 when two inputs are applied. If the inputs 010, 010, 100, 010, 100 are applied in sequence to the example SAR in Figure 27, the patterns generated by the SAR are 010, 011, 001, 110, 011 when the initial state is 000. Without detailed analysis, we can say that these patterns are close to random and the same patterns repeat before the SAR goes through a complete cycle. It is interesting to note that if only one input pattern is applied to an n-bit SAR, the SAR behaves like a maximum-length LFSR which generates 2^n-1 patterns. The pattern which is not generated depends on the polynomial used to implement the SAR and the input pattern applied. For example, the 3-bit SAR in Figure 27 does not generate the pattern 100 under the application of only one input pattern 010. Pattern 111 is not generated when the input pattern 011 is applied. However, when more than one input pattern is applied to an n-bit SAR, the SAR can generate all possible 2^n patterns. The next state for a given state of an SAR is distinct for each input pattern. For example, in Figure 28, the successor state of state 001 is state 110 for input pattern 010 and state 000 for input pattern 100. Similarly, the predecessor state of a given state is uniquely determined for an input pattern. For example, the predecessor state of state 001 is state 111 for input pattern 010 and state 011 for input pattern 100.

From the above observations, we conclude that the following three properties hold:

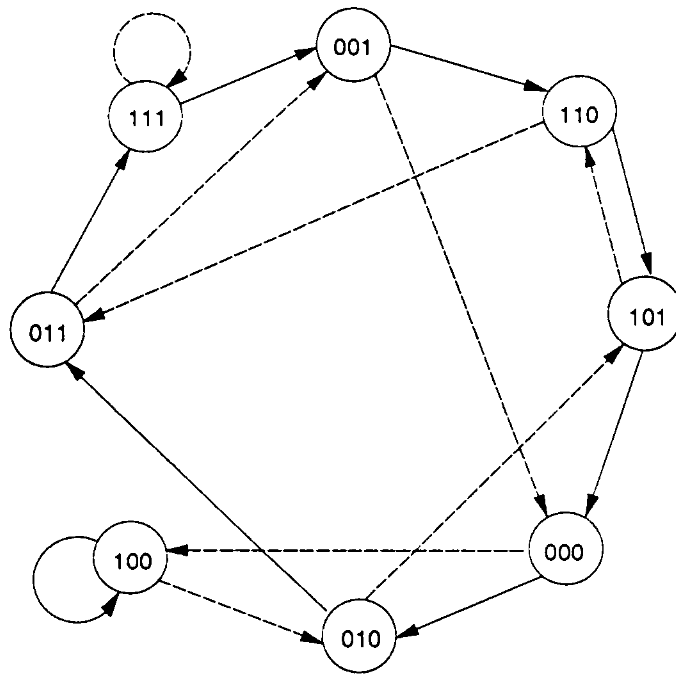
Property 1. The number of successor states of a given state of an SAR is equal to the number of input patterns of the SAR.

Property 2. The number of predecessor states of a given state of an SAR is equal to the number of input patterns of the SAR.



$$P(X) = X^3 + X^2 + 1$$

Figure 27. A 3-bit SAR.



Applied Input Patterns

Solid : $I_2I_1I_0$ = 010

Dashed : $I_2I_1I_0$ = 100

Figure 28. The state transition diagram of the 3-bit SAR in Figure 27 for two inputs.

Property 3. If the number of input patterns to an n-bit SAR is greater than one, the SAR can possibly generate all 2^n patterns.

Formal proofs of the properties are not given here, but are available in [55]. The above properties are necessary to investigate the randomness of the patterns generated by an SAR.

8.2.1 Randomness

Williams *et al.* showed that the behavior of SARs can be modeled as a Markov chain [56]. Let P_{ij} be the probability of state transition from state s_i to state s_j. Then, the state transition diagram of an n-bit SAR can be described by a 2^n X 2^n matrix, called transition probability matrix $P = [p_{ij}]$, $i, j=1,2, \ldots ,2^n$. For example, suppose that the probability of occurrence of input pattern 010 is 0.25 and the probability of input pattern 100 is 0.75 for the example SAR in Figure 27. Then, transition matrix P is obtained as:

$$P = \begin{pmatrix} 0 & 0 & 0.25 & 0 & 0.75 & 0 & 0 & 0 \\ 0.75 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0 \\ 0 & 0 & 0 & 0.25 & 0 & 0.75 & 0 & 0 \\ 0 & 0.75 & 0 & 0 & 0 & 0 & 0 & 0.25 \\ 0 & 0 & 0.75 & 0 & 0.25 & 0 & 0 & 0 \\ 0.25 & 0 & 0 & 0 & 0 & 0 & 0.75 & 0 \\ 0 & 0 & 0 & 0.75 & 0 & 0.25 & 0 & 0 \\ 0 & 0.25 & 0 & 0 & 0 & 0 & 0 & 0.75 \end{pmatrix}$$

Let π_j^k be the probability that an SAR is in state s_j after k clock cycles, i.e., after the application of k input patterns to the SAR. A state probability vector of an n-bit SAR is defined as:

$$\pi(k) = (\pi_0^k, \ \pi_1^k, \ \ldots, \ \pi_N^k), \quad \text{where N is } 2^n.$$

Then, $\pi(k)$ can be obtained using the initial probability vector $\pi(0)$ and the transition probability matrix P as given below:

$$\pi(k) \;=\; \pi(0)P^k.$$

For example, the state probability vectors of the SAR in Figure 28 under the initial state 000 can be obtained as follows:

$$\pi(1) \;=\; \pi(0)P \;=\; (1 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0)P$$

$$= (0 \;\; 0 \;\; 0 \;\; 0.25 \;\; 0 \;\; 0.75 \;\; 0 \;\; 0 \;\; 0)$$

$$\pi(2) \;=\; \pi(1)P \;=\; \pi(0)P^2 \;=\; \left(0 \;\; 0 \;\; \frac{9}{16} \;\; \frac{1}{16} \;\; \frac{3}{16} \;\; \frac{3}{16} \;\; 0 \;\; 0\right)$$

....

$$\pi(6) \;=\; \left(\frac{0.9375}{8} \;\; \frac{1.6875}{8} \;\; \frac{0.9375}{8} \;\; \frac{0.9375}{8} \;\; \frac{0.9375}{8} \;\; \frac{0.6875}{8} \;\; \frac{0.9375}{8} \;\; \frac{0.9375}{8}\right).$$

As can be observed from the above state probability vectors, each element of $\pi(k)$ approaches $\frac{1}{8}$ as k increases. This implies that the probability of appearance of each pattern for an SAR becomes identical after a sufficient number of clock cycles. We show this aspect using a known property of a Markov process.

A Markov chain is called *regular* if every state can be reached from every other state including the state itself regardless of the number of steps it takes. A transition probability matrix P of a Markov chain is called *doubly stochastic* if

$$\sum_k p_{ik} = \sum_k p_{kj} = 1, \quad \text{for all i and j.}$$

This means that the sum of each column and each row of P is equal to 1. The following property is known in the study of Markov processes [57].

If a Markov chain is regular and the transition probability matrix P is doubly stochastic, then

$$\lim_{k \to \infty} \pi(k) = \left(\frac{1}{N}, \frac{1}{N}, \ \ldots \ , \frac{1}{N}\right), \quad \text{where N is the number of state.}$$

From Property 1, 2, and 3 shown early in this section, the state diagram of an SAR is regular provided the number of input patterns is greater than one. Moreover, the transition probability matrix P of an SAR is doubly stochastic. Hence, we conclude the following theorem holds:

Theorem 1 (The Law of Large Numbers): For a given n-bit SAR, the probability of appearance of each pattern becomes $1/2^n$ after a sufficient number of clock cycles, provided the number of input patterns of the SAR is greater than one.

The above theorem is the Law of Large Numbers. However, the experimental results in the next section indicate that the number of clock cycles needed for making patterns random is small enough to say that the patterns generated by an n-bit SAR are random and they are uniformly distributed over the entire range from 0 to 2^n - 1.

8.2.2 Effectiveness

In the above section, we showed that every pattern of an SAR appears with equal probability during testing. However, this does not imply that the same patterns do not appear until every pattern appears at least once. This can be explained with a die tossing experiment. Suppose we toss a die five times. The probability of appearance of each number is $\frac{5}{6}$, but the same numbers may appear more than once in the trial. However, as long as the patterns generated by an SAR do not repeat too frequently, the SAR could be used effectively as pseudorandom test patterns. We investigate the

frequency of appearance in the following.

Let D_m be the number of different patterns among m patterns generated by an SAR. We define two probabilities $P_m(k)$ and $\lambda_m(k)$ using D_m as follows:

$$P_m(k) = \text{Prob}\,(D_m = k)$$

$$\lambda_m(k) = \text{Prob}\,(D_m = k \mid D_{m-1} = k-1).$$

$P_m(k)$ is the probability that k patterns are different among m patterns generated. $\lambda_m(k)$ is the conditional probability that the mth pattern generated does not belong to the previous k-1 different patterns among the m-1 patterns generated. Then, $P_m(k)$ can be expressed as follows:

$$P_m(k) = P_{m-1}(k-1)\lambda_m(k) + P_{m-1}(k)(1 - \lambda_m(k+1))$$

$$\text{for } 1<k<m \text{ and } 1<m$$

$$P_m(1) = P_{m-1}(1)(1 - \lambda_m(2)) \text{ for } k = 1 \text{ and } 2 \le m$$

$$P_m(m) = P_{m-1}(m - 1)\lambda_m(m) \text{ for } k = m \text{ and } 2 \le m.$$

The first term of $P_m(k)$ represents the case that m^{th} pattern generated does not belong to previous k-1 different patterns among m-1 patterns generated. The second term represents the opposite case that m^{th} pattern belongs to the previous k different patterns. The average number of different patterns among m patterns generated by an SAR is the expected value of D_m. The expected value of D_m is obtained as shown below (A formal derivation of the equation is given Appendix E):

$$E[D_m] = \sum_{k=1}^{m} kP_m(k)$$

$$= E[D_{m-1}] + \sum_{k=1}^{m-1} P_{m-1}(k)\lambda_m(k + 1).$$

The expected value of D_m is the sum of the expected value of D_{m-1} and the values obtained by generating one more pattern. It is difficult to find a closed form of the expected value $E[D_m]$ for general circuits since the conditional probability $\lambda_m(k)$ depends on the current state and the input pattern of the SAR. However, when all 2^n input patterns of an n-bit SAR are equiprobable, a closed form of $E[D_m]$ can be found. In the following theorem, we consider SARs in which all input patterns are equiprobable.

Theorem 2. If all 2^n inputs of an n-bit SAR are equiprobable, the average number of different patterns among the m patterns generated is

$$2^n\left(1-\left(1-\frac{1}{2^n}\right)^m\right).$$

Proof: Since all input patterns are equiprobable, the probability to transit from a given state to any state is equal to $1/2^n$. This means that all 2^n patterns can be generated in the next clock cycle with equal probability $1/2^n$. Hence, the probability that the mth pattern belongs to the previous k different patterns is $k/2^n$. Thus,

$$\lambda_m(k+1) = 1-\frac{k}{2^n}.$$

By substituting this into the equation for $E[D_m]$,

$$E[D_m] = E[D_{m-1}] - \frac{1}{2^n}\sum_{k=1}^{m-1} kP_{m-1}(k) + \sum_{k=1}^{m-1} P_{m-1}(k)$$

$$= E[D_{m-1}] - \frac{1}{2^n}E[D_{m-1}] + 1$$

$$= \left(1 - \frac{1}{2^n}\right)E[D_{m-1}] + 1.$$

Let $x(l)$ denote the sequence such that $x(l) = E[D_{m-1}]$, where l = m-1. Then, the above equation can be written as

$$x(l+1) = \left(1 - \frac{1}{2^n}\right)x(l) + 1.$$

The application of the z-transform to both sides of the above difference equation gives

$$zX(z) - zX(0) = \left(1 - \frac{1}{2^n}\right)X(z) + \frac{z}{1-z}.$$

Since the initial condition $x(0) = E[D_1] = 1$,

$$X(z) = \frac{z^2}{\left(z - 1 + \frac{1}{2^n}\right)(z-1)}.$$

Taking the inverse z-transform using the Cauchy integral theorem [58] gives

$$x(l) = 2^n\left(1 - \left(1 - \frac{1}{2^n}\right)^{l+1}\right).$$

Since $x(l) = E[D_{m-1}]$,

$$E[D_m] = 2^n\left(1 - \left(1 - \frac{1}{2^n}\right)^{m}\right).$$

Q.E.D.

The number of different patterns generated by an n-bit SAR for n = 4, 8, 16, 32 (obtained using Theorem 2) and that of a maximum-length n-bit Linear Feedback Shift Register (LFSR) are plotted in Figure 29. Since the scale of Figure 29 is too small to show the differences of $E[D_m]$ for the different values of n, only one graph is shown in Figure 29. In Figure 29, it can be seen that when m is smaller than 2^n, the number of different patterns of an SAR is close to m. This is the expected result since the chance of repeating the same patterns is small when a small number of patterns is generated. This fact also can be shown from Theorem 2 using an approximation rule. If $m \ll 2^n$, then

Average number of different patterns is

$$= 2^n\left(1 - \left(1-\frac{1}{2^n}\right)^m\right)$$

$$= 2^n\left(1 - \left(1-\frac{m}{2^n}\right)\right)$$

$$= m.$$

To measure the effectiveness of an SAR as a PPG let us define the *effectiveness* of a SAR as

Effectiveness of a MISR

$$= \frac{\text{number of different patterns generated}}{\text{total number of patterns generated}}$$

$$= \frac{E[D_m]}{m}.$$

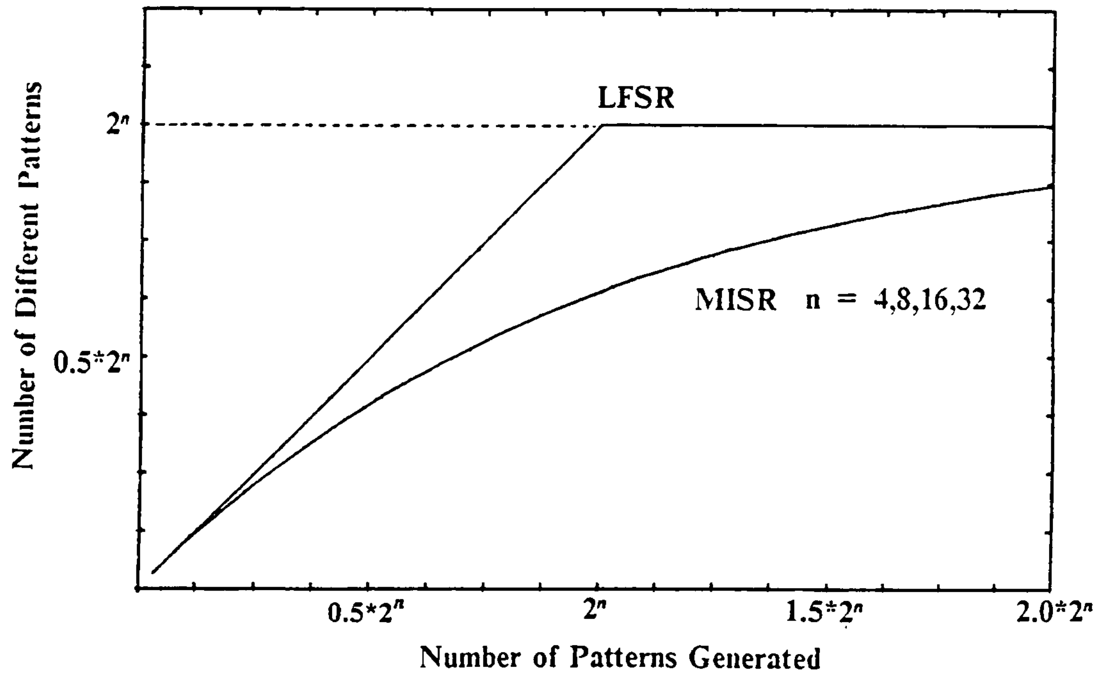


Figure 29. Number of different patterns generated by an n-bit SAR and an n-bit LFSR.

Using Theorem 2, the effectiveness of an n-bit SAR for equiprobable input patterns is plotted in Figure 30. As can be seen in the figure, a SAR can be effectively used as a PPG provided that a small portion of patterns (among all the possible patterns) are used. When the input patterns of an SAR are not equiprobable, the graph in Figure 30 is not applicable. However, the experimental results given in the next section strongly indicate that the number of different patterns generated by an SAR is relatively independent of the probabilities of input patterns. Hence, the same graph could be used to measure the effectiveness of an SAR.

8.3 Experimental Results

8.3.1 Randomness

As mentioned in the previous section, the patterns generated by an SAR become uniformly distributed after a sufficient number of clock cycles. However, in order to use an SAR as a PPG effectively, the number of clock cycles needed for the patterns to be uniformly distributed should be much smaller than the number of all possible patterns. In other words, the probability vector $\pi(k)$ of an n-bit SAR should be close to its final value

$$\pi = \left(\frac{1}{2^n}, \frac{1}{2^n}, \cdots, \frac{1}{2^n}\right) \quad \text{for small k.}$$

We define *settling time* of an n-bit SAR as the smallest value of k such that all the elements of $\pi(k)$ do not differ more than 5% of the final value $1/2^n$. Two major factors which determine the settling time of an SAR are the probabilities of occurrence of the input patterns and the length of the SAR. We conducted two different experiments to observe the effects of these two factors.

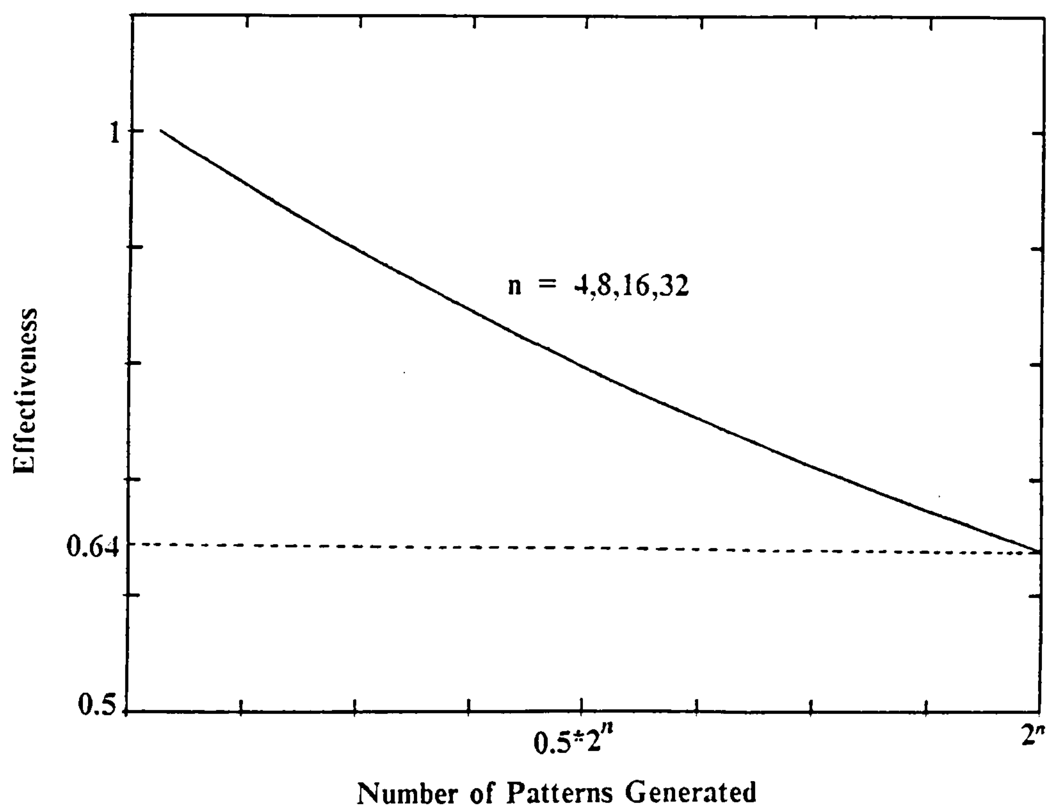


Figure 30. Effectiveness of an n-bit SAR as PPG.

The first experiment was performed to observe the effect of the probabilities of the input patterns of an SAR. In the experiment, we selected a certain number of input patterns randomly and assigned nonzero probabilities to the selected input patterns. The probabilities of the input patterns that are not selected are assigned to be 0. In assigning probabilities, we considered two cases, equiprobable input patterns and non-equiprobable input patterns. For example, when the number of input patterns is three, the probabilities of occurrence of the three patterns were given as 1/3 for the equiprobable case. For the non-equiprobable case, the probabilities of occurrence of input patterns were randomly assigned so that the sum of the probabilities is 1, e.g., 0.1, 0.2 and 0.7 for three input patterns. Using the selected input patterns and their probabilities, the transition probability matrix P was generated and the probability vector $\pi(k)$ was computed to find the settling time.

Experimental results for an 8-bit SAR are given in Table 6. For a given number of selected input patterns, we performed 20 tests. Input patterns and initial state are selected randomly in each experiment. The probabilities of selected input patterns for the case of non-equiprobable input patterns are also assigned randomly. The settling time given in Table 6 is the average of the 20 tests. The entry under the column heading "ratio" is the ratio of the settling time to the number of all possible patterns (256 in this experiment).

As shown in Table 6, the settling time decreases as the number of input patterns increases. When the number of input patterns is greater than 50, the settling time is less than four clock cycles for both the equiprobable and the non-equiprobable input patterns. The other observation made from the above experimental results is that the settling time for the equiprobable input patterns is shorter than for that of the non-equiprobable input patterns. Howver, the differences are negligible when compared to

Table 6. Settling Time of an 8-bit SAR.

No. of Input Patterns	Equiprobable Input Patterns		Non-equiprobable Input Patterns	
	Settling time	Ratio	Settling time	Ratio
2	8.00	0.0313	23.29	0.0910
3	11.25	0.0439	16.25	0.0635
5	8.30	0.0324	9.35	0.0365
10	6.05	0.0236	6.85	0.0268
20	5.00	0.0195	5.25	0.0205
50	4.00	0.0156	4.00	0.0156
100	3.00	0.0117	3.50	0.0137
150	3.00	0.0117	3.15	0.0123
200	2.45	0.0096	3.00	0.0117
250	2.00	0.0078	3.00	0.0117

the number of all possible patterns of the SAR.

The second experiment was performed using SARs of various lengths. In the experiment, the number of input patterns is fixed to three since it gives the longest settling time for the equiprobable input patterns of an 8-bit SAR (refer to Table 6). As in the first experiment, 20 tests were performed for each SAR. In each test, initial state, input patterns, and probabilities were randomly chosen. Experimental results are shown in Table 7.

As shown in Table 7, the settling times of SARs with the length of nine or greater are less than 4% of the number of all possible patterns. Moreover, as the length of SAR increases, the ratio of the settling time to the number of all possible patterns is decreased. For a 14-bit SAR, the ratio for the non-equiprobable patterns is only 0.14%. It should be noted that the results in Table 7 are obtained when the number of input patterns is three. In general, the number of output patterns of a circuit, i.e., the number of input patterns of an SAR, is greater than three. Therefore, the settling time will be further reduced.

In summary, the settling time of an SAR is decreased as the number of input patterns increases. The ratio of the settling time to the number of all possible patterns of an SAR is decreased substantially as the length of the SAR is increased. The experimental results indicate that the settling time of an SAR is far smaller than the number of all possible patterns of the SAR. This implies that most of the patterns generated by an SAR are random.

8.3.2 Effectiveness

An estimate of the number of different patterns generated by an SAR was discussed in Section 8.2.2. We experimented with a 16-bit SAR for various CLBs. The

Table 7. Settling Times for SARs with Various Lengths.

Length of SAR	Equiprobable Input Patterns		Non-equiprobable Input Patterns	
	Settling time	Ratio	Settling time	Ratio
6	9.60	0.1500	14.05	0.2195
7	10.40	0.0813	15.55	0.1215
8	11.25	0.0439	16.25	0.0634
9	12.70	0.0248	16.40	0.0320
10	13.75	0.0134	17.65	0.0172
11	14.15	0.0069	18.55	0.0091
12	14.70	0.0036	19.65	0.0048
13	16.00	0.0020	21.65	0.0026
14	16.60	0.0010	22.80	0.0014

16-bit SAR was chosen because we believe the number of all the possible patterns for this SAR (i.e., 65,536) is large enough to obtain meaningful statistical data. The objectives of the experiment were

1. to test the accuracy of the equation given in Theorem 2 which estimates the number of different patterns for equiprobable input patterns, and
2. to estimate the number of different patterns for non-equiprobable input patterns.

A block diagram showing the experimental procedure is given in Figure 31. The PPG in the figure was implemented using a maximum-length LFSR. The size of the LFSR is equal to the number of inputs of the test circuit. All the test circuits used in the experiments have 16 outputs. While applying pseudorandom patterns to the test circuit, the outputs of the SAR were observed and the number of different patterns generated by the SAR was counted.

Five test circuits, an 8x8 multiplier, a 16-bit ALU, a 16-bit adder, a 4x16 decoder and a randomly generated PLA, were used in the experiment. The first three circuits (multiplier, ALU, and adder) can generate all possible output patterns, 2^{16} different output patterns. The last two circuits (the decoder and the PLA) can generate only limited numbers of output patterns, e.g., 16 output patterns for the decoder. The probability of the appearance of an output pattern depends on the function of the circuit. Hence, the probability of an input pattern for the SAR is a function of the test circuit. For the last two test circuits, the probabilities of input patterns for the SAR are very different depending on the input pattern applied to the test circuit. For example, during testing of the decoder, only 16 input patterns for the SAR have the probability 1/16 each. The rest of the input patterns have the probability 0. This is because the decoder can produce only 16 different output patterns.

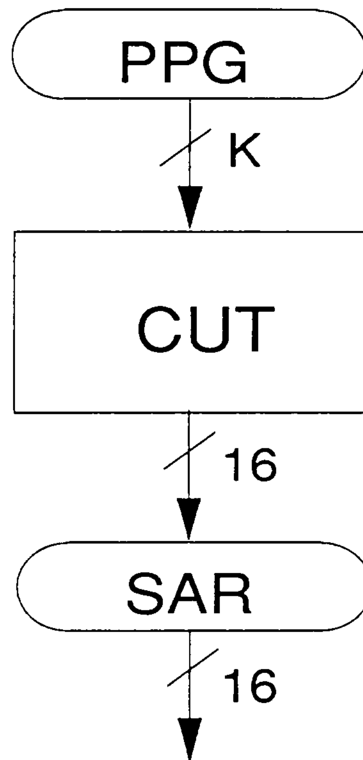


Figure 31. Experiment for effectiveness of a 16-bit SAR as PPG.

The experimental results for the five circuits are shown in Table 8. Each circuit has two entries for a given number of patterns applied during testing. The top entry is the number of different patterns generated by the SAR while the bottom entry is the effectiveness (%) of the SAR as a PPG for the circuit. As shown in Table 8, the numbers of different patterns obtained from the experiments are close to their estimated numbers for all five circuits. The effectiveness of the SAR for each circuit does not differ more than 2.0% from the estimated value. It is interesting that the probabilities of input patterns of the SAR does not significantly reduce the effectiveness of the SAR. This is highly desirable since it implies that an SAR generates a certain number of different random patterns regardless of the characteristics of the circuit which drives it. From the experiments, the following two observations can be made:

Observation 1. The equation given in Theorem 2 accurately estimates the number of different patterns for any number of patterns generated by the SAR.

Observation 2. The probabilities of input patterns of an SAR do not affect the number of different patterns generated by the SAR.

We were unable to obtain a closed form equation estimating the number of different patterns generated by an SAR for non-equiprobable input patterns. However, the experimental results strongly indicate that the equation for the case of equiprobable input patterns is also valid for non-equiprobable input patterns. We conclude this section by providing the following conjecture:

Conjecture: The average number of different patterns among m patterns generated by an n-bit SAR is

Table 8. Experimental Results for the Number of Different Patterns Generated by 16-bit SAR.

No. of Test Patterns	Multiplier	ALU	Adder	Decoder	PLA	Estimates
10K	9315 (93.15)	9265 (92.65)	9264 (92.64)	9309 (93.09)	9301 (93.01)	9274 (92.74)
20K	17256 (86.28)	17256 (86.28)	17300 (86.50)	17350 (86.75)	17376 (86.88)	17236 (86.18)
30K	24072 (80.24)	24093 (80.31)	24150 (80.50)	24438 (81.46)	24126 (80.72)	24072 (80.24)
40K	29964 (74.91)	29904 (74.76)	30052 (75.13)	29904 (75.87)	30352 (75.88)	29940 (74.85)
50K	34925 (69.85)	34975 (69.95)	35065 (70.13)	35545 (71.09)	34010 (68.02)	34975 (69.95)
60K	39252 (65.42)	39372 (65.62)	39576 (65.69)	40098 (66.83)	39198 (65.33)	39300 (65.50)

$$2^n\left(1-\left(1-\frac{1}{2^n}\right)^m\right).$$

8.4 A Case Study

In order to investigate the practicality of using the patterns of an SAR as pseudorandom test patterns, we experimented with the design shown in Figure 10 for various BIST schemes. Figure 32 shows a portion of the design used in the experiment. To reduce the computation time, the size of data path is reduced by half (from 32 to 16 bits). The Multiplier and the ALU are implemented with 74-series TTL gates. The Multiplier is implemented using SN74274, SN74275, and SN7483 gates. The ALU is implemented using SN74181 and SN74182 gates. The 8x8 multiplier and the 16-bit ALU in Figure 32 have 880 and 354 gates, respectively. The number of single stuck-at fault classes is 1,255 for the multiplier and 574 for the ALU.

Suppose that the multiplier and the ALU are to be tested using a BIST scheme. Several different BIST schemes can be applied to the circuit as discussed below.

a) Two session testing

A block diagram for this case is shown in Figure 2 in Chapter 2. The multiplier and the ALU are tested one at a time. An extra 22-bit PPG is added to drive the left 16 inputs, five control inputs, and the carry-in of the ALU.

b) Single signature testing

A block diagram for this case is shown in Figure 26. The multiplier and the ALU are directly cascaded through the register positioned between the two modules. An extra 22-bit PPG is also needed here for the ALU. During testing, the outputs of the

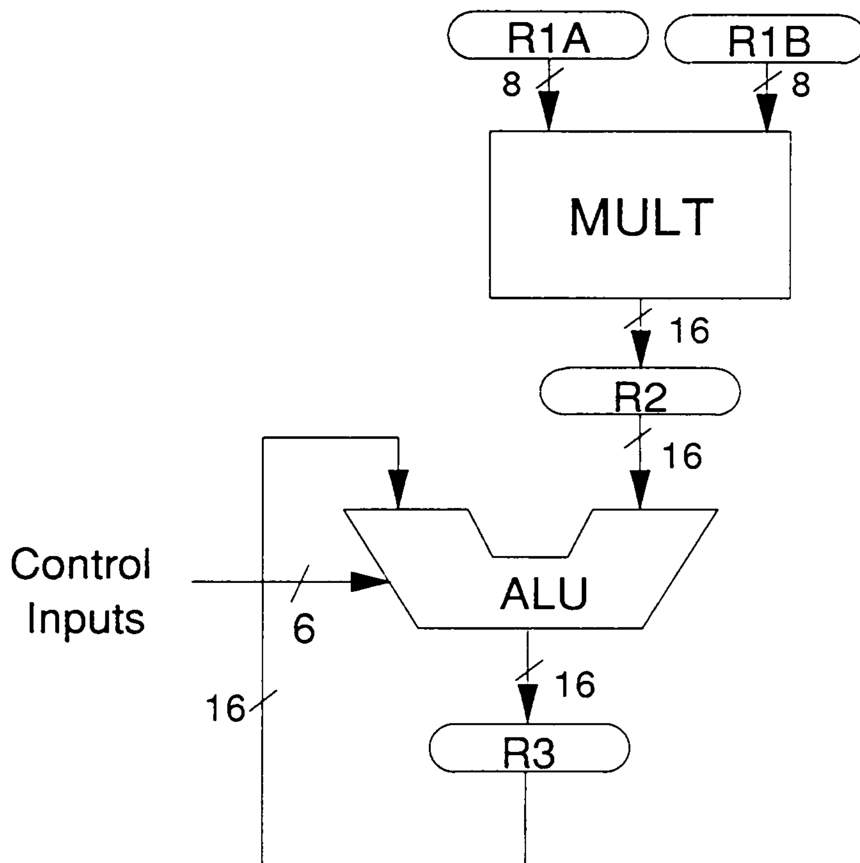


Figure 32. An example circuit for fault simulation experiment.

multiplier are fed to the ALU. Only the outputs of the ALU are collected by the signature register for verification of the test. This scheme needs only one testing session.

*c) **Proposed scheme I***

R2 is extended to be a 38-bit SAR as shown in Figure 33. The patterns generated by the extended SAR are applied to test the ALU. This scheme also requires only one testing session.

*d) **Proposed scheme II***

This scheme takes advantage of the original architecture of the chip. The patterns of the SAR R3 (for the ALU) are fed back to the ALU. An extra 6-bit PPG is needed for the control inputs and carry-in of the ALU. A single testing session is needed.

We simulated the above four different self-testing schemes. For each method, 20 fault simulations were performed using distinct seed numbers for the PPGs. We did not consider signature aliasing errors in the simulations. Experimental results are shown in Table 9. For the two session testing the number of test patterns was obtained by adding the number of test patterns required to test the multiplier and that for the ALU.

From Table 9, it can be seen that the two session testing schem and both the proposed schemes I and II achieve 100% fault coverage for all single stuck-at faults. Single signature testing reaches a fault coverage saturatation only at 64.5%, which is well below 100%. We observed that on the average 95% of the undetected faults for the single signature testing reside in the multiplier and the remaining 5% in the ALU. This was found to be because of the poor observability of the multiplier.

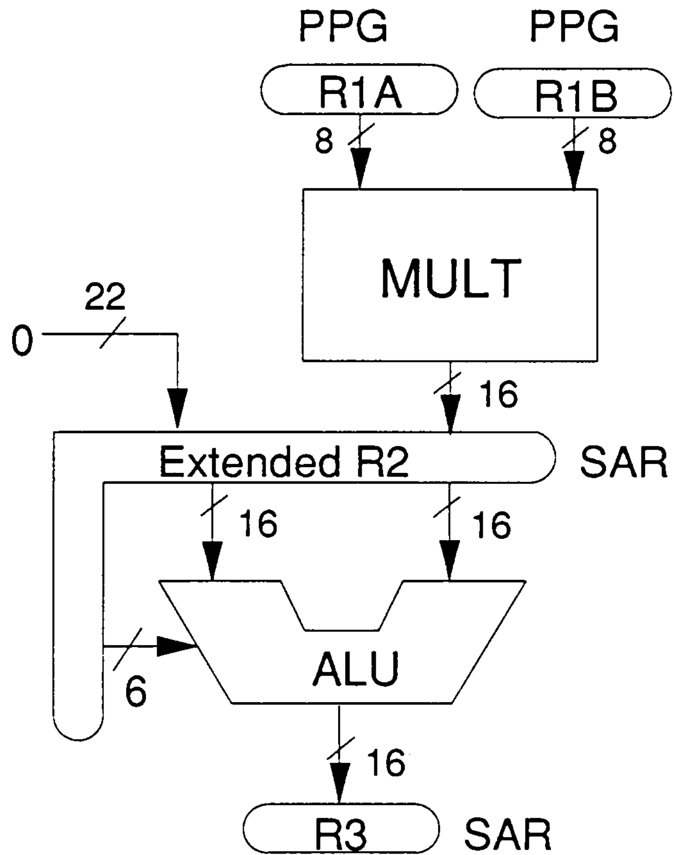


Figure 33. Proposed Scheme I.

Table 9. Fault Simulation Result for Various Self-Testing Schemes.

No. of Test Patterns	BILBO Scheme	Single Signature Testing	Proposed Scheme I	Proposed Scheme II
Average	2,177	> 3000	1,457	1,378
Minimum	830	-	634	721
Maximum	3,619	-	2,531	2,212
Fault Coverage (%)	100.0	64.5	100.0	100.0

Both of the proposed schemes I and II significantly reduce the number of test patterns necessary when compared with that of the two session testing scheme. The proposed scheme I achieves a 33% reduction in the test size while a 37% reduction is obtained for the scheme II. We observed that the 38-bit SAR for the proposed scheme I seldom repeats the same patterns. This is because the test size is small when compared with all of the possible patterns for the SAR. For the scheme II, the inputs applied to the ALU depend on the current state of SAR R3. Hence, this scheme does not guarantee that the patterns generated by the SAR are uniformly distributed. (The probability transition matrix P is not doubly stochastic.)

In summary, the experimental results indicate that an SAR can be effectively used as a PPG regardless of the circuit that drives the SAR, provided the number of test patterns is relatively small when compared with the number of possible states of the SAR.

Chapter 9

Conclusions

9.1 BIDES

9.1.1 Contribution

The central theme in developing BIDES was the consideration of design objectives, i.e., hardware overhead and testing time, in a design system for BILBO-based BIST. The consideration of design objectives spawned the requirement that the system must have the capability to explore various BIST structures for a given circuit. Most of the existing CAD tools for BIST design lack this capability. This requirement caused the problem of BIST design to be formulated as a search, i.e., find a BIST structure which can meet the design objectives. A typical search technique for design problems, an iterative process of regeneration and evaluation, was applied naturally.

The next issue was the implementation of BIDES. The problem of BIST design is not an optimization problem; the conventional space-time trade-off exists. Therefore, the problem cannot be solve algorithmically. Moreover, knowledge about the BIST structures, especially variations of BILBO techniques such as feedback testing and cascade testing, are required in order to incorporate such BIST structures into a design. These characteristics of the problem led to the use of knowledge-based expert system approach.

Since there are several methods to implement a knowledge-based system, the most suitable method for BIST design had to be chosen. BIDES could have been implemented with only if-then production rules. However, implementation of the entire

system with the if-then production rules is not very effective since knowledge representation is very complicated. Moreover, it is relatively difficult to update the knowledge in if-then production rules since the coded knowledge cannot be easily traced. For this reason, a more abstract method of knowledge representation was sought. This resulted in defining the abstraction hierarchy in BIST design. Using the abstraction hierarchy, hierarchical planning can be applied fairly easily. The ease of implementation is attributed to the fact that the entire problem can now be divided into several simplified problems within each abstraction level. Each such problem has a limited scope. The concept of test structure played an especially vital role in the implementation of BIDES.

Using the abstraction hierarchy, knowledge about BIST design methods was represented with operators defined on abstraction levels, i.e., procedural knowledge. This type of knowledge representation made it easy to understand the design procedure taken. Furthermore, this methodology in turn leads to an easily modifiable system. The new knowledge can be added or modified by simply defining new operators.

In summary, the major contribution of the dissertation is that BIDES is capable of considering a wide variety of BIST structures to be applied to a given circuit. This capability was able to be implemented easily by using hierarchical planning in conjunction withe the abstraction hierarchy. The use of hierarchical planning also provides flexibility for updating the knowledge-base.

9.1.2 Future Directions

BIDES also has some limitations. The first contention issue is the testing of special structures such as PLAs, RAMs, and ROMs. The self-testing of these structures requires peculiar techniques rather than just allocating testing resources. It means that we need another type of knowledge-based system for self-testable design of special

structures. Other researchers are continuing to work on automatic design of self-testable PLAs and memory elements. Once this research comes to fruition, the issue is how to integrate two types of CAD tools for self-testable design, one for special structures and the other for random logic such as BIDES. One problem in integrating two types of CAD tools is the difference in levels of abstraction used in the tools. The self-testable design of special structures requires the circuit modification at the gate level. On the other hand, self-testable design of random logic can be performed at the register transfer level as is done with BIDES. This means that the insertion of DFT features must be considered at the register transfer level as well as the gate level.

The second issue is the measures for evaluating a BIST implementation. In BIDES, the hardware overhead is estimated using a transistor count. However, The silicon area necessary for any the extra signal paths is not taken into account. We can obtain only rough estimates for the hardware overhead with the transistor count. Since hardware overhead is one of the major factors in BIST, the actual silicon area should be estimated rather than the just transistor count.

The final issue is the scan-path organization. In BIDES, it is assumed that all flip-flops are placed in the single scan-path. However, the large number of flip-flops in the single scan-path can make the testing time fairly long. In this case, the use of multiple scan-paths must be considered. Moreover, some flip-flops may not be able to be placed in the scan-path if they are in critical system paths. In this case, we need to consider the use of partial scan-paths.

9.2 Study on Cascade Testing

An analytical study on cascade testing is also a large part of the contribution of this dissertation. Through this analytical study, it is shown that a signature analysis register can be effectively used as a pseudorandom pattern generator when two

combinational logic modules are cascaded. We have shown that the patterns produced by a signature analysis register become uniformly distributed. As shown in the experimental results, the number of clock cycles needed for generating uniformly distributed patterns, i.e., the settling time, is small compared to the number of all possible patterns. The settling time depends on the probabilities of appearance of input patterns in signature analysis registers. However, it has been shown in several examples that the effect of probabilities is negligible. According to the experimental results, the ratio of the settling time to the number of all possible patterns is less than 1% for signature analysis registers with a length of 10 bits or more. Moreover, the ratio is decreased substantially as the length of the SAR is increased. Considering the number of inputs of typical data path modules in VLSI circuits, the patterns generated by signature registers can be considered as random patterns.

Another characteristic we have shown is that the patterns generated by signature registers are rarely repeated when the number of patterns is small compared to the number of all possible patterns. The behavior of the number of different patterns generated by signature analysis registers is relatively independent of the probabilities of appearance of input patterns. In conclusion, we have shown that cascade testing can be used to reduce the overall testing time because of the two characteristics: randomness of patterns and the rare repetition of the same patterns.

One of the future research problem in this area is to prove the conjecture provided for the non-equiprobable inputs. Through experiments, it has been shown that the formula for estimating number of different patterns generated by an SAR is still valid for the non-equiprobable input case. Recall that the formula was driven for the equiprobable inputs. However, this has not been proven analytically. Another interesting problem is the use of a signature analysis register as a test pattern generator in

feedback testing. In cascade testing, the patterns generated by a signature analysis register are not dependent on the circuit whose signature is collected by the signature analysis register. However, in feedback testing, the contents of the signature analysis register are again used for testing the module itself. Using fault simulation experiments, we have shown that feedback testing is a feasible BIST scheme. However, we still need an analytical study similar to the one performed for cascade testing.

Bibliography

1. T.W. Williams and K.P. Parker, "Design for Testability - A Survey," *IEEE Trans. Computers*, Vol. C-32, No. 1, Jan. 1982, pp. 2-15.

2. T.W. Williams and K.P. Parker, "Testing Logic Networks and Design for Testability," *Computer*, Oct. 1979, pp. 9-21.

3. E.J. McCluskey, "Built-In Self-Test Techniques," *IEEE Design and Test of Computers*, Apr. 1985, pp. 21-28.

4. E.J. McCluskey, "Built-In Self-Test Structures," *IEEE Design and Test of Computers*, Apr. 1985, pp. 29-36.

5. A.C. Parker and S. Hayati, "Automating the VLSI Design Process Using Expert System and Silicon Compilation," *Proceedings of IEEE*, Vol. 75, No. 6, Jun. 1987, pp. 777-785.

6. K. Kim, D.S. Ha and J.G. Tront, "On Using Signature Register as Pseudorandom Pattern Generator," *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems*, Vol. 7, No. 8, Aug. 1988, pp. 919-928.

7. F.P. Beucler and M.J. Manner, "HILDO : Highly Integrated Logic Device Observer," *VLSI Design*, Jun. 1984, pp. 88-96.

8. J.W. Gannet, "Self-Testing by Integrated Feedback (STIF)," *in VLSI Electronics, N.G. Einspruch (Ed.)* , Academic Press, 1986, pp. 107-110.

9. E.B. Eichelberger and T.W. Williams, "A Logic Design Structure for LSI Testability," *Jour. of Design Automation and Fault Tolerant Computing*, Vol. 2, May 1978, pp. 165-178.

10. J.H. Stewart, "Future Testing of Large LSI Circuit Cards," *Proc. Semiconductor Test Symposium*, 1977, pp. 6-17.

11. H. Ando, "Testing VLSI with Random Access Scan, *Proc. COMPCON Spring*, 1980, pp. 50-52.

12. P.H. Bardell, W.H. McAnney and J. Savir, *Built-In Test for VLSI*, John Wiley and Sons, 1987.

13. C.K. Chin and E.J. McCluskey, "Test Length for Pseudorandom Testing," *IEEE Trans. Computers*, Vol. C-36, No. 2, Feb. 1987, pp. 252-256.

14. K.D. Wagner, C.K. Chin and E.J. McCluskey, "Pseudorandom Testing," *IEEE Trans. Computers*, Vol. C-36, No. 3, Mar. 1987, pp-332-343.

15. H.J. Nadig, "Signature Analysis - Concepts, Examples and Guidelines," *Hewlett-Packard Jour.*, May 1977, pp. 15-21.

16. J.P. Hayes, "Transition Count Testing of Combinational Logic Circuit," *IEEE Trans. Computers*, Vol. C-25, Jun. 1976, pp. 613-620.

17. J.Savir, "Syndrome Testable Design of Combinational Circuits," *IEEE Trans. Computers*, Vol. C-29, Jun. 1980, pp. 442-451.

18. B. Koenemann, J. Mucha and G. Zwiehoff, "Built-In Test for Complex Digital Integrated Circuits," *IEEE Jour. Solid-State Circuits*, Vol. SC-15, No. 3, Jun. 1980, pp. 315-318.

19. Y.M. El-Zig, "S^3 : VLSI Self-Testing using Signature Analysis and Scan-Path Techniques," *Proc. ICCAD*, 1984, pp. 89-101.

20. P.H. Bardell and W.H. McAnney, "Self-Testing of Multi-Chip Logic Modules," *Proc. Int'l. Test Conf.*, 1982, pp. 200-204.

21. D. Komonystky, "LSI Self-Testing using LSSD and Signature Analysis," *Proc. Int'l. Test Conf.*, 1982, PP. 414-424.

22. N. Yamaguch, "A Self-Testing Method for Modular Structured Logic VLSIs," *Proc. ICCAD*, 1984, pp. 99-101.

23. P.P. Fasang, "BIDCO, Built-In Digital Circuit Observer," *Proc. Int'l. Test Conf.*, pp. 261-266.

24. P.W. Horstmann, "A Knowledge-Based System Using Design for Testability Rules," *Proc. 14th Int'l. Symposium on Fault Tolerant Computing*, 1984, pp. 278-284.

25. V.D. Agrawal, S.K. Jail and D.M. Singer, "A CAD System for Design for Testability," *VLSI Design*, Oct. 1984, pp. 46-54.

26. P. Camurati *et Al.*, "ESTA : An Expert System for DFT Rule Verification," *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems*, Vol. 7., No. 11. Nov. 1988. pp. 1172-1180.

27. H.S. Fung and S. Hirschhorn, "An Automatic DFT System for the Silc Silicon Compiler," *IEEE Design and Test of Computers*, Feb. 1986, pp. 45-57.

28. H.S. Fung, S. Hirshhorn and R. Kullarni, "Design for Testability in a Silicon Compilation Environment," *Proc. 22nd Design Automation Conf.*, 1985. pp. 190-196.

29. M.A. Samad and J.A.B. Fortes, "DEFT - A Design for Testability Expert System," *Proc. Fall Joint Computer Conf.*, 1986. pp. 899-908.

30. IBM Federal Systems Divisions, *Overview, Logic Entry, Design for Testability - Master Image Designer's Guide*, 1985.

31. M.A. Samad and J.A.B. Fortes, "Explanation Capability in DEFT," *Proc. Int'l. Test Conf.*, 1986, pp. 954-963.

32. M.S. Abadir and M.A. Breuer, "A Knowledge-Based System for Designing Testable VLSI Chips," *IEEE Design and Test of Computers*, Aug. 1985, pp. 56-68.

33. M.A. Jones and K. Baker, "An Intelligent Knowledge-Based System Tool for High-Level BIST Design," *Proc. Int'l Test Conf.*, 1985. pp. 743-746.

34. J. Granacki, D. Knapp and A. Parker, "The ADAM Advanced Design Automation System : overview, Planner, and Natural-language Interface," *Proc. 22nd Design Automation Conf.*, 1985.

35. K. Kim, J.G. Tront and D.S. Ha, "Automatic Insertion of BIST Hardware using VHDL," *Proc. 25th Design Automation Conf.*, 1988, pp. 9-15.

36. R. Oxman and J.S. Gero, "Using an Expert System for Design Diagnosis and Design Synthesis," *Expert Systems*, Vol. 4., No. 1, Feb. 1987, pp. 4-15.

37. E. Charniak and D. McDermott, *Introduction to Artificial Intelligence,* Addison Wesley, 1985.

38. N.C. Rowe, *Artificial Intelligence Through Prolog*, Prentice Hall, 1988.

39. M.P. Georgeff and A.L. Lansky, "Procedural Knowledge," *Proceedings of IEEE*, Vol. 74, No. 10, Oct. 1986, pp. 1383-1397.

40. P.R. Cohen and E.A. Feigenbaum, *The Handbook of Artificial Intelligence*, William Kaufmann, Inc., 1982.

41. M. Shadad, "An Overview of VHDL Language Technology," *Proc. 23rd Design Automation Conf.*, 1986, pp. 320-326.

42. K.Lin *et al.*, "The TMS320 Family of Digital Signal Processor," *Proceedings of IEEE*, Vol. 75, No. 9, 1987, pp. 1143-1159.

43. Design Automation Technical Committee of the Computer Society of the IEEE, *IEEE Standard VHDL Language Reference Manual*, IEEE, Inc., 1988.

44. G. Cabodi, P. Camurati and P. Prinetto, "The Use of Prolog Specification and Verification of Easily Testable Designs," *Proc. 16th Int'l. Symposium on Fault Tolerant Computing*, 1986, pp. 390-395.

45. M.S. Abadir and M.A. Breuer, "Test Schedules for VLSI Circuits Having Built-In Test Hardware," *IEEE Trans. Computers*, Vol. C-35, No. 4, Apr. 1986, pp. 361-367.

46. C.R. Kime and K.K. Saluja, "Test Scheduling in Testable VLSI Circuits," *Proc. Int'l. Test Conf.*, 1982. pp. 406-412.

47. N. Christofides, *Graph Theory: An Algorithmic Approach*, Academic Press, 1975.

48. J.J. Ohletz, T.W. Williams and J.P. Mucha, "Overhead in Scan and Self-Testing Designs," *Proc. Int'l. Test Conf.*, 1987. pp. 460-470.

49. K. Kim, J.G. Tront and D.S. Ha, "On Hardware Overhead in CMOS BIST Designs," *Proc. Southeastcon*, 1989.

50. A.V. Aho, J.E. Hopcroft and J.D. Ullman, *Data Structures and Algorithms*, Addison Wesley, 1982.

51. J.Kalinowki, A. Albicki and J. Beausang, "Test Control Signal Distribution in Self-Testing VLSI Circuits," *Proc. ICCAD*, 1986, pp. 60-63.

52. J.L Carter, "The Theory of Signature Testing for VLSI," *Proc. ACM Symposium on Theory of Computing*, 1982, pp. 66-76.

53. A. Krasniewski and S. Pliarski, "Circular Self-Test: A Low-Cost BIST Techniques for VLSI Circuit," *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems*, Vol. 8, No. 1, Jan. 1989, pp. 46-55.

54. L.T. Wang and E.J. McCluskey, "Built-In Self-Test for Sequential Machines," *Proc. Int'l. Test Conf.*, 1987, pp. 334-341.

55. S.W. Golumb, *Shift Register Sequences*, Holden-Day, 1967.

56. T.W. Williams *et al.*, "Comparison of Aliasing Error for Primitive and Non-primitive Polynomials," *Proc. Int'l. Test Conf.*, 1986, pp. 282-288.

57. H.M. Taylor and S. Karlin, *An Introduction to Stochastic Modeling*, Academic Press, 1984.

58. A.V. Oppenheim and R.W. Schafer, *Digital Signal Processing*, Prentice Hall, 1975.

Appendix A
Prolog Description of Hardware

In this appendix, the Prolog description of the example design in Figure 10 is provided. The description is translated from the VHDL description in Figure 11.

```
module(acc,reg,
        [inport(d,alu,output,data,bit_vector(0,31)),
         inport(cp,clk,clk,clock,bit)],
        [outport(q,bus_data,bus_data,data,bit_vector(0,15)),
         outport(q,para_shift,input,data,bit_vector(0,31)),
         outport(q,alu,in1,data,bit_vector(0,31))]).

module(alu,clb,
        [inport(in1,acc,q,data,bit_vector(0,31)),
         inport(in2,mux2,output,data,bit_vector(0,31)),
         inport(cntl,ctl_alu,ctl_alu,cntl,bit_vector(0,3))],
        [outport(output,acc,d,data,bit_vector(0,31))]).

module(ar0,reg,
        [inport(d,bus_data,bus_data,data,bit_vector(0,15)),
         inport(cp,clk,clk,clock,bit)],
        [outport(q,bus_data,bus_data,data,bit_vector(0,15))]).

module(ar1,reg,
        [inport(d,bus_data,bus_data,data,bit_vector(0,15)),
         inport(cp,clk,clk,clock,bit)],
        [outport(q,bus_data,bus_data,data,bit_vector(0,15))]).
```

```
module(barr_shift,clb,
        [inport(input,bus_data,bus_data,data,bit_vector(0,15)),
         inport(cntl,ctl_ba,ctl_ba,cntl,bit_vector(0,3))],
        [outport(output,mux2,in1,data,bit_vector(0,31))]).

module(bus_data,bus,
        [inport(bus_data,para_shift,output,data,bit_vector(0,15)),
         inport(bus_data,acc,q,data,bit_vector(0,15)),
         inport(bus_data,ar0,q,data,bit_vector(0,15)),
         inport(bus_data,ar1,q,data,bit_vector(0,15))],
        [outport(bus_data,mult,mult2,data,bit_vector(0,15)),
         outport(bus_data,barr_shift,input,data,bit_vector(0,15)),
         outport(bus_data,ar0,d,data,bit_vector(0,15)),
         outport(bus_data,ar1,d,data,bit_vector(0,15)),
         outport(bus_data,treg,d,data,bit_vector(0,15))]).

module(clk,primary_in,[],
        [outport(clk,preg,cp,clock,bit),
         outport(clk,acc,cp,clock,bit),
         outport(clk,ar0,cp,clock,bit),
         outport(clk,ar1,cp,clock,bit),
         outport(clk,treg,cp,clock,bit)]).

module(ctl_alu,primary_in,[],
        [outport(ctl_alu,alu,cntl,cntl,bit_vector(0,3))]).

module(ctl_ba,primary_in,[],
        [outport(ctl_ba,barr_shift,cntl,cntl,bit_vector(0,3))]).

module(ctl_mux2,primary_in,[],
        [outport(ctl_mux2,mux2,select,cntl,bit)]).

module(ctl_pa_sh,primary_in,[],
        [outport(ctl_pa_sh,para_shift,cntl,cntl,bit_vector(0,3))]).
```

```
module(mult,clb,
        [inport(mult1,treg,q,data,bit_vector(0,15)),
         inport(mult2,bus_data,bus_data,data,bit_vector(0,15))],
        [outport(output,preg,d,data,bit_vector(0,31))]).

module(mux2,mux,
        [inport(in1,barr_shift,output,data,bit_vector(0,31)),
         inport(in2,preg,q,data,bit_vector(0,31)),
         inport(select,ctl_mux2,ctl_mux2,cntl,bit)],
        [outport(output,alu,in2,data,bit_vector(0,31))]).

module(para_shift,clb,
        [inport(input,acc,q,data,bit_vector(0,31)),
         inport(cntl,ctl_pa_sh,ctl_pa_sh,cntl,bit_vector(0,3))],
        [outport(output,bus_data,bus_data,data,bit_vector(0,15))]).

module(preg,reg,
        [inport(d,mult,output,data,bit_vector(0,31)),
         inport(cp,clk,clk,clock,bit)],
        [outport(q,mux2,in2,data,bit_vector(0,31))]).

module(treg,reg,
        [inport(d,bus_data,bus_data,data,bit_vector(0,15)),
         inport(cp,clk,clk,clock,bit)],
        [outport(q,mult,mult1,data,bit_vector(0,15))]).
```

Appendix B

Syntax for Frame Representation of Objects

As described in Chapter 4, the structure of a design is represented using the abstraction hierarchy. The abstraction hierarchy has four classes of objects: test structure, CUT, Register and Port. Each object is represented using frames. In this appendix, the syntax for the frame representation of each object is provided.

In representing a frame, triples are used to represent slot values instead of the usual two-argument predicates. The syntax is

value(<object>, <slot>, <value>).

It means that the <slot> of the <object> has the <value>. The name of each object which is used as the frame name is generated internally. The name of a frame is not identical with the original name of the object which is given in the VHDL description. When an object is referenced in another object, the object is referred to by the frame name.

B.1 Test Structure Frame

frame(<frame id>).

value(<frame id>, is_a, test_struct).

value(<frame id>, cut, <cut frame id>).

value(<frame id>, test_scheme, <test scheme>).

value(<frame id>, test_time, <no. of test patterns>).

value(<frame id>, tpg, <list of port-register pair>).

value(<frame id>, sar, <list of port-register pair>).

value(<frame id>, sartpg, <list of port-register pair>).

value(<frame id>, unalloc_ports, <list of port>).

The slot "is_a" just represents that the frame is a test structure frame. The "cut" slot represents the CUT tested in the test structure, which is referred to by <cut frame id>. The value of the "test_scheme" slot is given according to the testing schemes used to test the CUT. There are three possible values for the "testing scheme": *pseudorandom*, *feedback* and *cascade*. The "tpg" slot represents the allocation of a test pattern generator for input ports of the CUT. The value of the "tpg" slot is a list of port-register pairs. A port-register pair is a list which consists of port and allocated tpg, e.g., [inport1, reg5]. If an input port does not have any available resource for its test pattern generator, the register in the port-register pair is specified as "not_avail" which implies that the test pattern generator for the input port is not available. If an input port cannot be allocated due to the constraints of pseudorandom testing, the word "unallocatable" is used in the position of register. The "sar" slot represents the allocation of signature analyzer for the output ports of the CUT. The value of the "sar" slot is given in the exactly same manner as the value the "tpg" slot is specified. The "sartpg" slot is used for the cascade testing or feedback testing in which a signature analyzer is used as a test pattern generator. The "unalloc_port" slot is asserted when there is a port which cannot be allocated due to either the constraints or no available resource. The value of the slot is the list of ports which cannot be allocated.

B.2 Circuit Frame

```
frame(<frame id>).
value(<frame id>, is_a, circuit).
value(<frame id>, name, <name of CLB>).
value(<frame id>, test_time,<test time>).
value(<frame id>, inputs, <list of input ports>).
value(<frame id>, outputs, <list of output ports>).
```

A circuit frame represents a combinational logic block in a design, which is referred as a CUT in a test structure frame. The "is_a" slot indicates that the frame is a circuit frame. The value of the "name" slot is the original name of the circuit, which is given in the VHDL description. The "test_time" slot represents the number of required pseudorandom patterns to test the circuit. The value <test time> is given by the designer. The value of the "inputs" slot is the list of input ports of the circuit. An element of the list is the name of a port frame. Similarly, the value of the "outputs" slot is the list of output ports of the circuit. The list also consists of frames for output ports.

B.3 Input Port Frame

```
frame(<frame id>).
value(<frame id>, is_a, <"input_port" or "primary_in">).
value(<frame id>, a_part_of, <circuit frame id>).
value(<frame id>, port_name, <name of port>).
value(<frame id>, size, <size of the port>).
```

value(<frame id>, sources, <list of candidate register frames for TPG>).

value(<frame id>, tpg, <allocated register frame as the TPG>).

value(<frame id>, sar_tpg, <allocated register frame as the SAR TPG>).

An input port frame represents an input port of a combinational logic block. If type of the port is indicated by the "is_a" slot. If the port is a primary input port, the value is given as "primary_in". Otherwise, the value of the slot is specified as "input_port". The circuit frame which contains the port is referred by the "a_part_of" slot. The original name and size of the port are given in the "port_name" slot and "size" slot, respectively. In the "sources" slot, the register frames which can be used as test pattern generator for the port are given. There exists a transparent path between a candidate register and the port. If there is no register which can be used as the test pattern generator, the value of the "sources" slot is specified as "not_avail". For the primary input ports, the value of the slot is specified as "external". When a register is allocated as a TPG, the "tpg" slot is filled with the allocated register. If the test patterns for the port are generated by a signature analyzer, i.e., cascade testing or feedback testing, then the "sar_tpg" slot is filled instead of "tpg" slot.

B.4 Output Port Frame

frame(<frame id>).

value(<frame id>, is_a, <"output_port" or "primary_out">).

value(<frame id>, a_part_of, <circuit frame id>).

value(<frame id>, port_name, <name of the port>).

value(<frame id>, size, <size of the port>).

value(<frame id>, sinks, <list of candidate register frames for SAR>).

value(<frame id>, sar, <allocated register frame as the SAR>).

The output port frame is similar to the input port frame. An output port frame has the "sinks" slot instead of "tpg" slot in an input port frame, which represents the candidate registers for the SAR of the output port. After, a register is allocated as the SAR of the port, the "sar" slot is filled. The value of the "sar" slot will be given as "not_avail" if the port has no available resource for the SAR.

B.5 Register Frame

frame(<frame id>).

value(<frame id>, is_a, register).

value(<frame id>, name, <name of the register>).

value(<frame id>, inport, <name of the input port>).

value(<frame id>, outport, <name of the output port>).

value(<frame id>, gds, <global degree of sharing of the register>).

value(<frame id>, tpg_of, <port which uses the register as TPG>).

value(<frame id>, sar_of, <port which uses the register as SAR>).

A register frame has also "is_a" and "name" slot like other frames. Since a register has a single input port and single output port, the name of input port and output port are stored directly in "inport" and "outport" slot, respectively. The "gds" slot represents the global degree of sharing of register which is mentioned in Chapter 5. Whenever the register is allocated as TPG or SAR, a "tpg_of" slot is asserted. The value of the slot is the port which uses the register as TPG. Similarly, a "sar_of" slot is asserted whenever the register is allocated as a signature analyzer.

Appendix C

Restrictions on VHDL Modeling

The system takes a VHDL description of a design as an input. Then, the VHDL description is translated into a Prolog description. Since VHDL allows various modeling styles, some restrictions are given in modeling the design using VHDL. As mentioned in Chapter 4, the design must be described using strict structural modeling, i.e., component declaration followed by component instantiation, at the register transfer level. In order to obtain a processable Prolog description of the design, the following rules must be followed in structural modeling.

1. **Modeling Style**: Modeling must be done using strict structural modeling which consists of type declaration, entity declaration, architectural body. The architectural body must be modeled with component declaration, signal declaration and component instantiation statements.

2. **Signal Declaration**: In the system, signals are classified into four types: control, data, bus and clock. Since signal types must be recognized from the description, the following prefixes must be used in declaring signals in entity declaration and architectural body:

   control: CTL (e.g., CTL_ALU)

   bus: BUS (e.g., BUS_DATA)

   clock: CLK (e.g., CLK_SYSTEM)

   The signals without the predefined prefix will be considered as data.

3. **Component Declaration**: There are four types of components in the system: combinational logic block, register, multiplexer, and memory. In declaring components in the architectural body, the component must be one of the five types. No other type is allowed, e.g., an individual gate cannot be declared. The component must also be declared using the following prefixes:

combination logic block:	CLB (e.g., CLB_ALU)
register:	REG (e.g., REG_ACC)
multiplxer:	MUX (e.g., MUX_16)
memory:	MEM (e.g., MEM_RAM)

Appendix D

Examples

In this appendix, test runs on five example circuits are provided.

D.1 Example 1

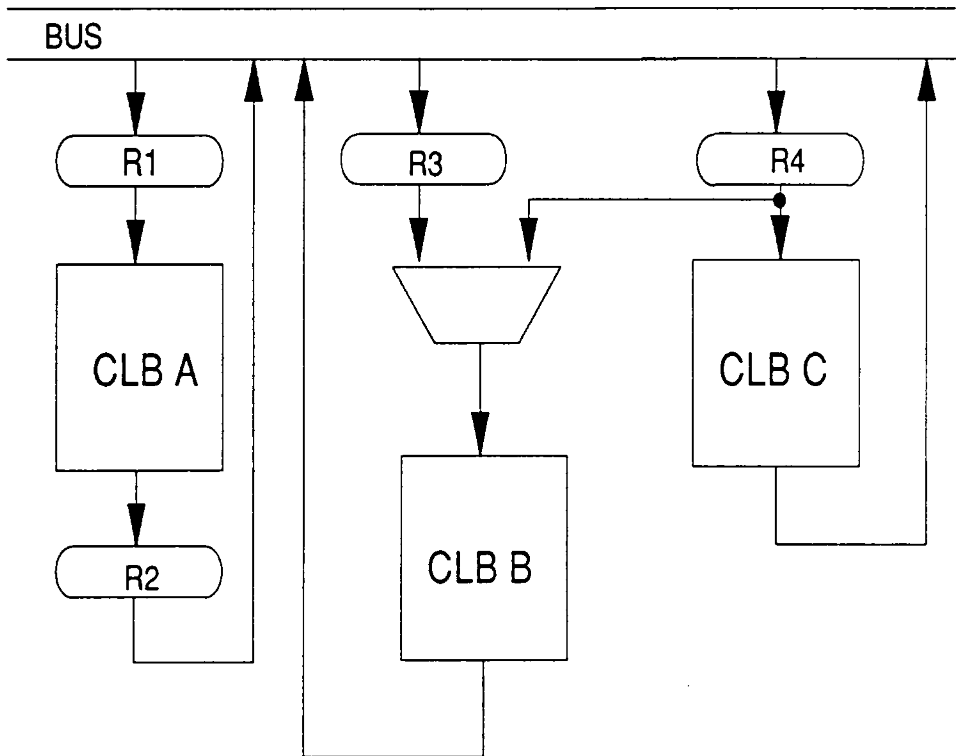
The example circuit shown in Figure 34 is used for the first example. Three different BIST structures are implemented by BIDES. Each BIST structure is shown below.

D.1.1 Structure 1

The BIST structure shown below is the result of initial state generation.

- **Testing Resource Allocation**

CLB	TPG	SAR	Test Scheme
CLB A	R1	R2	Pseudorandom
CLB B	R4	R1	Pseudorandom
CLB C	R4	R1	Pseudorandom



(All data paths are 16-bits wide.)

Figure 34. Example circuit 1.

- **Hardware Overhead**

Component	Modification (Size)	Hardware Overhead (TRs)
R1	BILBO (16)	328
R2	SAR (16)	318
R3	SCAN (16)	64
R4	TPG (16)	100
Total		810

- **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB A	1234
Session 2	CLB B	465
Session 3	CLB C	879
Total		2578

D.1.2 Structure 2

Structure 2 is generated from Structure 1 by eliminating R2 in order to reduce hardware overhead. The total hardware overhead is reduced from 810 TRs to 784 TRs.

- **Testing Resource Allocation**

CLB	TPG	SAR	Test Scheme
CLB A	R1	R4	Pseudorandom
CLB B	R4	R1	Pseudorandom
CLB C	R4	R1	Pseudorandom

- **Hardware Overhead**

Component	Modification (Size)	Hardware Overhead (TRs)
R1	BILBO (16)	328
R2	SCAN (16)	64
R3	SCAN (16)	64
R4	TPG (16)	100
Total		784

• **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB A	1234
Session 2	CLB B	465
Session 3	CLB C	879
Total		2578

D.1.3 Structure 3

Structure 3 is generated from Structure 1 in order to reduce testing time. CLB A is tested in parallel with CLB B by cascade testing scheme.

• **Testing Resource Allocation**

CLB	TPG	SAR	Test Scheme
CLB A	R1 (SARTPG)	R2	Cascade
CLB B	R4	R1	Pseudorandom
CLB C	R4	R1	Pseudorandom

• **Hardware Overhead**

Component	Modification (Size)	Hardware Overhead (TRs)
R1	SAR (16)	318
R2	SAR (16)	318
R3	SCAN (16)	64
R4	TPG (16)	100
Total		800

• **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB C	879
Session 2	CLB A CLB B	1245.76 465
Total		2124.76

D.2 Example 2

The second example circuit is shown in Figure 35. Three BIST structures are generated by BIDES.

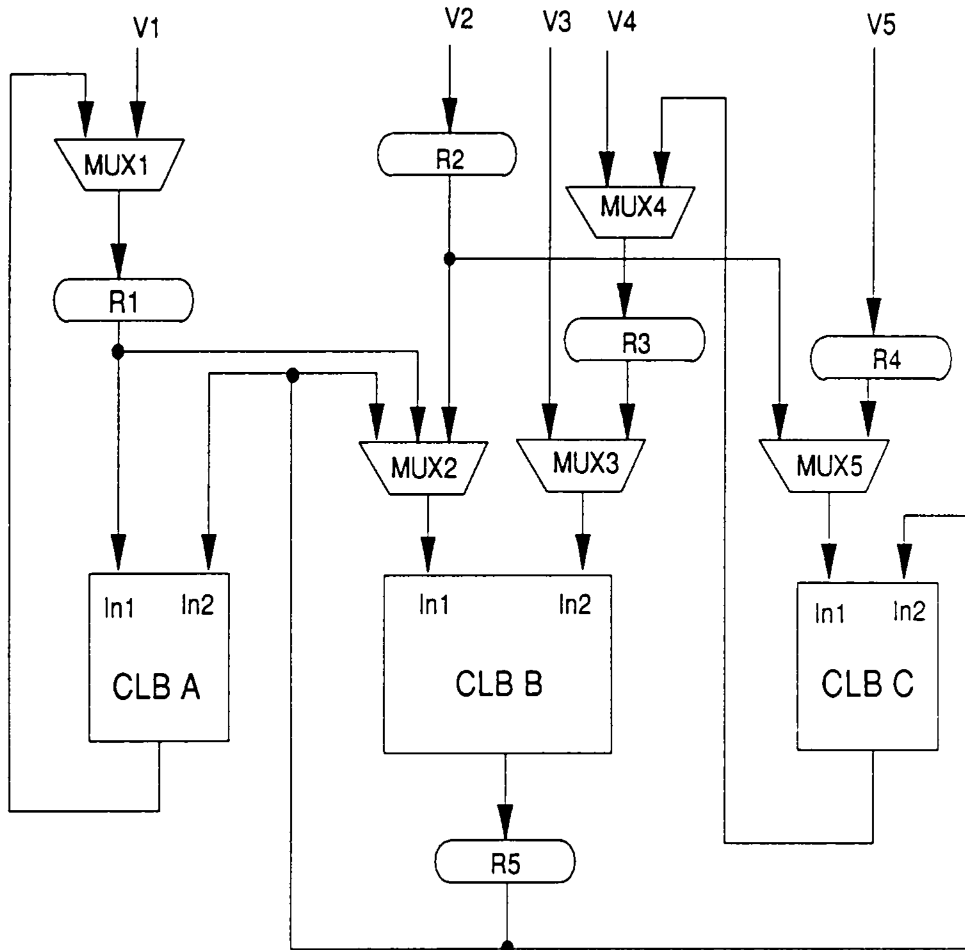
D.2.1 Structure 1

The BIST structure shown below is the result of initial state generation.

- **Testing Resource Allocation**

CLB	TPG (In1)	TPG (In2)	SAR	Test Scheme
CLB A	R3	R5	R1	Pseudorandom
CLB B	R1	R3	R5	Pseudorandom
CLB C	R2	R5	R3	Pseudorandom

As shown above, R3 is allocated as TPG for the port In1 of CLB A. Since there is no existing path from R3 to the port, an extra 2 X 1 MUX EXMUX is added to the port In1 of CLB A.



(All data paths are 16-bits wide.)

Figure 35. Example circuit 2.

• **Hardware Overhead**

Component	Modification (Size)	Hardware Overhead (TRs)
R1	BILBO (16)	328
R2	TPG (16)	100
R3	BILBO (16)	328
R4	SCAN (16)	64
R5	BILBO (16)	328
EXMUX	2 X 1 (16)	64
Total		1212

• **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB A	2578
Session 2	CLB B	3584
Session 3	CLB C	1129
Total		7291

D.2.2 Structure 2

Structure 2 is generated from Structure 1 by eliminating R2 in order to reduce hardware overhead. However, total hardware overhead is increased due to the modification of MUX5 into 4 X 1 MUX.

- **Testing Resource Allocation**

CLB	TPG (In1)	TPG (In2)	SAR	Test Scheme
CLB A	R3	R5	R1	Pseudorandom
CLB B	R1	R3	R5	Pseudorandom
CLB C	R1	R5	R3	Pseudorandom

- **Hardware Overhead**

Component	Modification (Size)	Hardware Overhead (TRs)
R1	BILBO (16)	328
R2	SCAN (16)	64
R3	BILBO (16)	328
R4	SCAN (16)	64
R5	BILBO (16)	328
EXMUX	2 X 1 (16)	64
MUX5	4 X 1 (16)	128
Total		1304

• **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB A	2578
Session 2	CLB B	3584
Session 3	CLB C	1129
Total		7291

D.2.3 Structure 3

Structure 3 is generated from Structure 1 in order to reduce testing time. CLB A is tested in parallel with CLB B by cascade testing scheme. The port In2 of CLB A is stimulated by patterns generated by R5 which collects the signature for CLB B.

• **Testing Resource Allocation**

CLB	TPG (In1)	TPG (In2)	SAR	Test Scheme
CLB A	R3	R5 (SARTPG)	R1	Cascade
CLB B	R2	R3	R5	Pseudorandom
CLB C	R2	R5	R3	Pseudorandom

- **Hardware Overhead**

Component	Modification (Size)	Hardware Overhead (TRs)
R1	SAR (16)	318
R2	TPG (16)	100
R3	BILBO (16)	328
R4	SCAN (16)	64
R5	BILBO (16)	328
EXMUX	2 X 1 (16)	64
Total		1202

- **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB C	1129
Session 2	CLB A	2630.06
	CLB B	3584
Total		4713

D.3 Example 3

The third example circuit is shown in Figure 36. Three BIST structures are generated by BIDES.

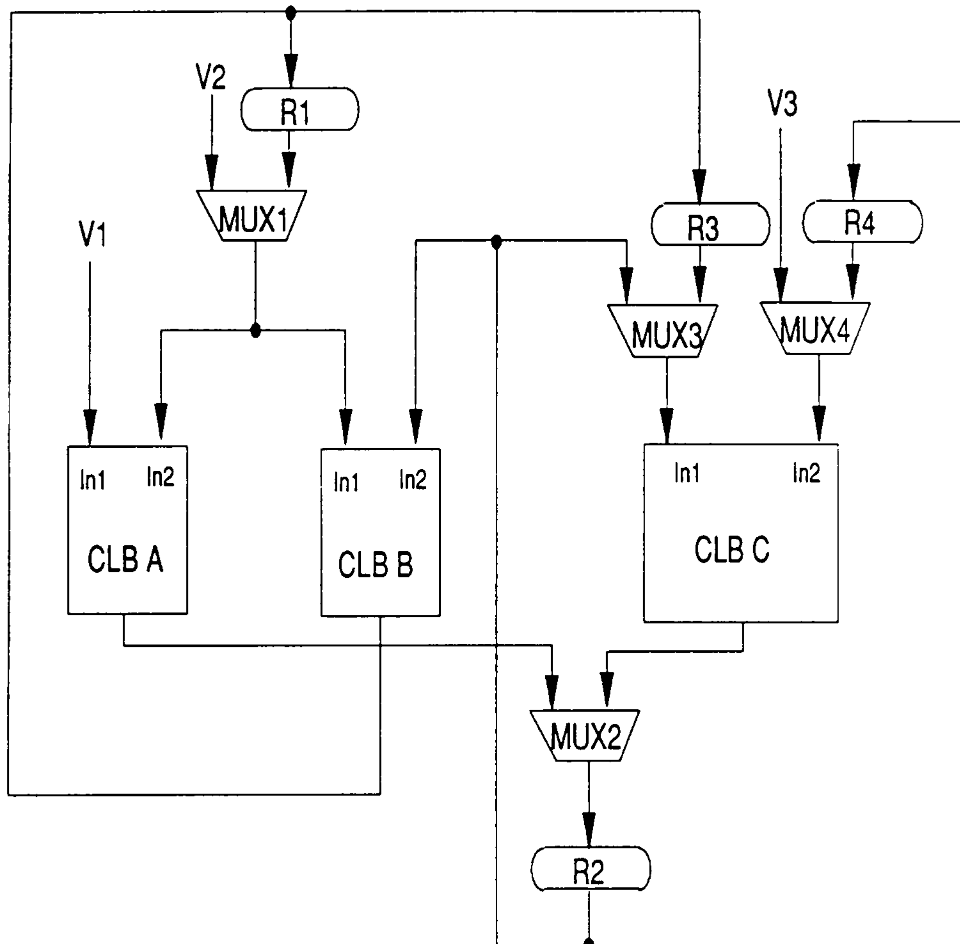
D.3.1 Structure 1

The BIST structure shown below is the result of initial state generation.

- **Testing Resource Allocation**

CLB	TPG (In1)	TPG (In2)	SAR	Test Scheme
CLB A	external TPG	R1	R2	Pseudorandom
CLB B	R1	R2	R3	Pseudorandom
CLB C	R3	R4	R2	Pseudorandom

Since the port In1 of CLB A is a primary input, the use of external TPG is assumed by BIDES.



(All data paths are 16-bits wide.)

Figure 36. Example circuit 3.

• **Hardware Overhead**

Component	Modification (Size)	Hardware Overhead (TRs)
R1	TPG (16)	100
R2	BILBO (16)	328
R3	BILBO (16)	328
R4	TPG (16)	100
Total		856

• **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB A	569
Session 2	CLB B	384
Session 3	CLB C	1210
Total		2163

D.3.2 Structure 2

When BIDES is asked to reduce hardware overhead, Structure 2 is generated from Structure 1. Hardware overhead is reduced by eliminating R4 and CLB C is

tested using the feedback testing scheme. The port In2 of CLB C is stimulated by the patterns generated by R2 which is collecting the signature of CLB C. Total hardware overhead is reduced from 856 TRs to 820 TRs.

• **Testing Resource Allocation**

CLB	TPG		SAR	Test Scheme
	(In1)	(In2)		
CLB A	external TPG	R1	R2	Pseudorandom
CLB B	R1	R2	R3	Pseudorandom
CLB C	R3	R2 (SARTPG)	R2	Feedback

• **Hardware Overhead**

Component	Modification (Size)	Hardware Overhead (TRs)
R1	TPG (16)	100
R2	BILBO (16)	328
R3	BILBO (16)	328
R4	SCAN (16)	64
Total		820

• **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB A	569
Session 2	CLB B	384
Session 3	CLB C	> 1210
Total		> 2163

Notice that testing time of CLB C is greater than 1210 because of the use of feedback testing. BIDES cannot estimate testing time of feedback testing.

D.3.3 Structure 3

BIDES generated Structure 3 from Structure 1 when it is asked to reduce testing time. CLB B is tested by cascade testing scheme in parallel with CLB C. The port In2 of CLB B is stimulated by R2 which collects the signature of CLB C. Total testing time is reduced from 2163 to 1779.

- **Testing Resource Allocation**

CLB	TPG (In1)	TPG (In2)	SAR	Test Scheme
CLB A	external TPG	R1	R2	Pseudorandom
CLB B	R1	R2 (SARTPG)	R3	Cascade
CLB C	R3	R4	R2	Pseudorandom

- **Hardware Overhead**

Component	Modification (Size)	Hardware Overhead (TRs)
R1	TPG (16)	100
R2	SAR (16)	318
R3	BILBO (16)	328
R4	TPG (16)	100
Total		846

• **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB A CLB B	569 385.13
Session 2	CLB C	1210
Total		1779

D.4 Example 4

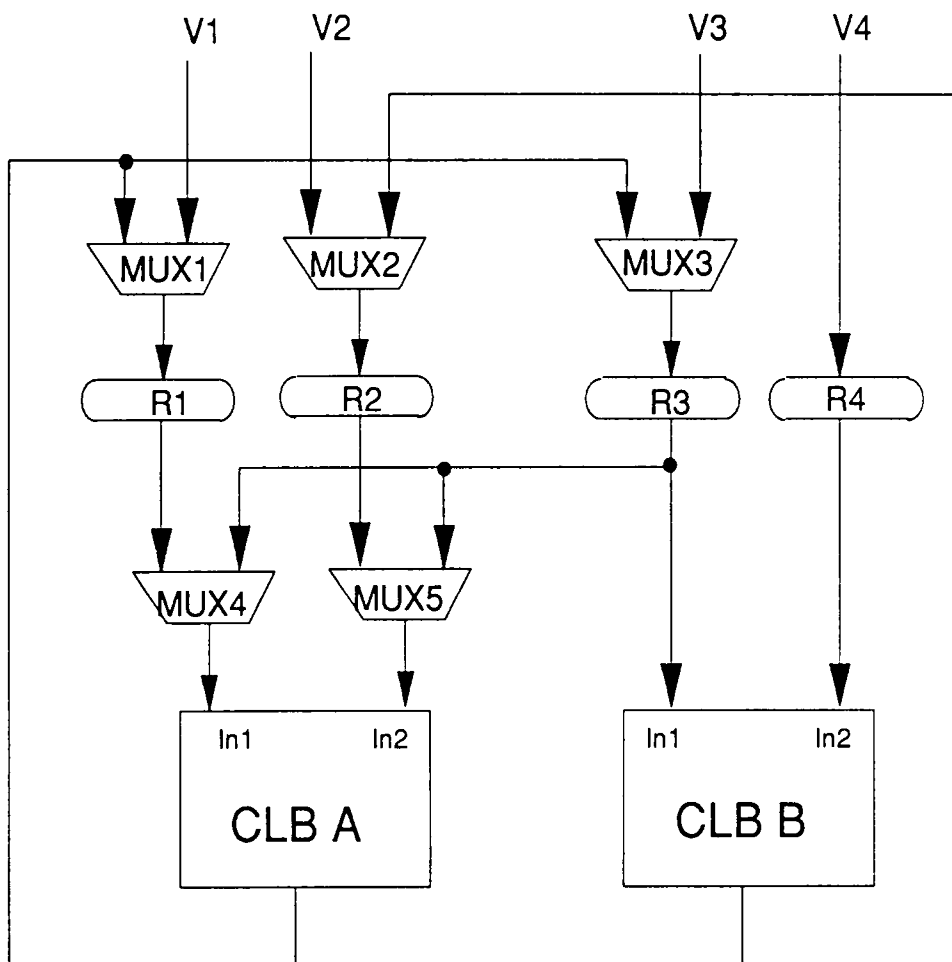
The fourth example circuit is shown in Figure 37. Three BIST structures are generated by BIDES.

D.4.1 Structure 1

The BIST structure shown below is the result of initial state generation.

• **Testing Resource Allocation**

CLB	TPG		SAR	Test Scheme
	(In1)	(In2)		
CLB A	R3	R2	R1	Pseudorandom
CLB B	R3	R4	R2	Pseudorandom



(All data paths are 16-bits wide.)

Figure 37. Example circuit 4.

• **Hardware Overhead**

Component	Modification (Size)	Hardware Overhead (TRs)
R1	SAR (16)	318
R2	BILBO (16)	328
R3	TPG (16)	100
R4	TPG (16)	100
Total		846

• **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB A	2122
Session 2	CLB B	759
Total		2881

D.4.2 Structure 2

When BIDES is asked to reduce hardware overhead, Structure 2 is generated from Structure 1. Hardware overhead is reduced by eliminating R1 and CLB A is

tested using the feedback testing scheme. The port In1 of CLB A is stimulated by the patterns generated by R3 which is collecting the signature of CLB A. Total hardware overhead is reduced from 846 TRs to 820 TRs.

- **Testing Resource Allocation**

CLB	TPG (In1)	TPG (In2)	SAR	Test Scheme
CLB A	R3 (SARTPG)	R2	R3	feedback
CLB B	R3	R4	R2	Pseudorandom

- **Hardware Overhead**

Component	Modification (Size)	Hardware Overhead (TRs)
R1	SCAN (16)	64
R2	BILBO (16)	328
R3	BILBO (16)	328
R4	TPG (16)	100
Total		820

• **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB A	> 2122
Session 2	CLB B	759
Total		> 2881

Notice that testing time of CLB A is greater than 2122 because of the use of feedback testing. BIDES cannot estimate testing time of feedback testing.

D.4.3 Structure 3

BIDES generates Structure 3 from Structure 1 based upon the request for reducing testing time. As shown below, two CLBs are tested in parallel with the cascade testing of CLB A. The port In2 of CLB A is stimulated by R2 which collects the signature of CLB B. Total testing time is reduced from 2881 to 2157.01

• **Testing Resource Allocation**

CLB	TPG		SAR	Test Scheme
	(In1)	(In2)		
CLB A	R3	R2 (SARTPG)	R1	Cascade
CLB B	R3	R4	R2	Pseudorandom

• **Hardware Overhead**

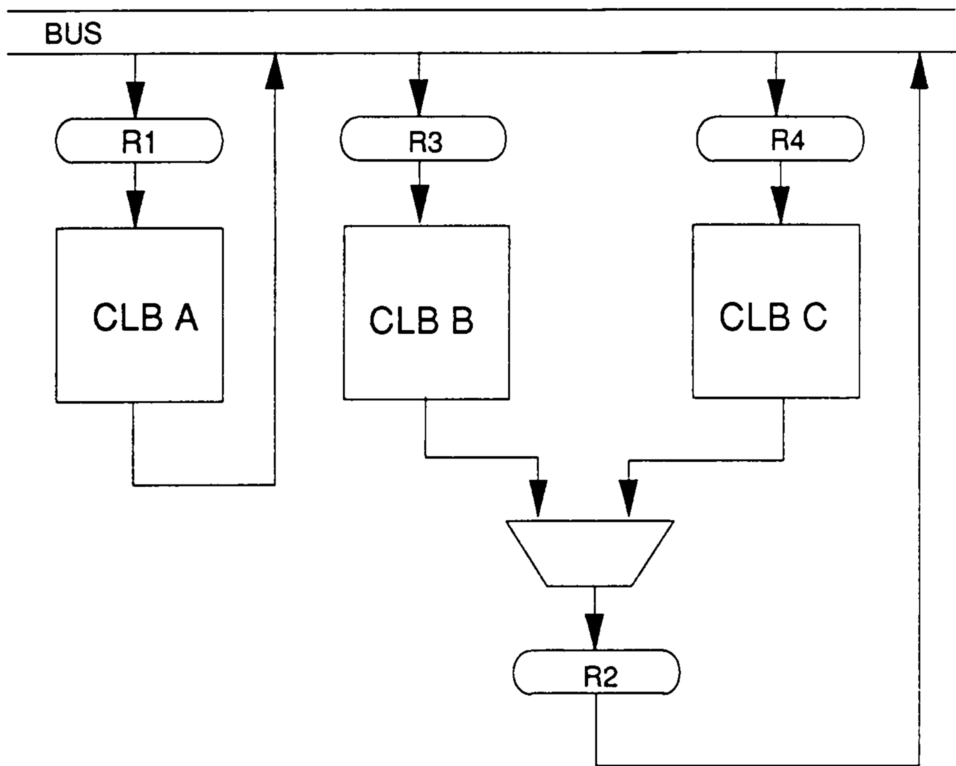
Component	Modification (Size)	Hardware Overhead (TRs)
R1	SAR (16)	318
R2	SAR (16)	318
R3	TPG (16)	100
R4	TPG (16)	100
Total		836

• **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB A	2157.01
	CLB B	759
Total		2157.01

D.5 Example 5

The example circuit shown in Figure 38 is used for the last example. Four different BIST structures are implemented by BIDES. Each BIST structure is shown below.



(All data paths are 16-bits wide.)

Figure 38. Example circuit 5.

D.5.1 Structure 1

The BIST structure shown below is the result of initial state generation.

- **Testing Resource Allocation**

CLB	TPG	SAR	Test Scheme
CLB A	R1	R3	Pseudorandom
CLB B	R3	R2	Pseudorandom
CLB C	R4	R2	Pseudorandom

- **Hardware Overhead**

Component	Modification (Size)	Hardware Overhead (TRs)
R1	TPG (16)	100
R2	SAR (16)	318
R3	BILBO (16)	328
R4	TPG (16)	100
Total		846

• **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB A CLB C	528 824
Session 2	CLB B	336
Total		1160

D.5.2 Structure 2

Structure 2 is generated from Structure 1 by eliminating R3 in order to reduce hardware overhead. The total hardware overhead is reduced from 846 TRs to 810 TRs. Testing time becomes greater than 1688 from 1160 of Structure 1.

• **Testing Resource Allocation**

CLB	TPG	SAR	Test Scheme
CLB A	R1	R4	Pseudorandom
CLB B	R2 (SARTPG)	R2	feedback
CLB C	R4	R2	Pseudorandom

• **Hardware Overhead**

Component	Modification (Size)	Hardware Overhead (TRs)
R1	TPG (16)	100
R2	SAR (16)	318
R3	SCAN (16)	64
R4	BILBO (16)	328
Total		810

• **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB A	528
Session 2	CLB B	> 336
Session 3	CLB C	824
Total		> 1688

Since CLB B is tested by feedback testing, the testing time for CLB B cannot be estimated. We need three separate test sessions and it gives the testing time which is greater than 1688.

D.5.3 Structure 3

Structure 3 is generated from Structure 2 by eliminating R4 in order to reduce hardware overhead. The total hardware overhead is reduced from 810 TRs to 764 TRs. Note that all three CLBs are tested using feedback testing, and each CLB needs separate test session due to the bus sharing. Thus, the total testing time will be greater than the sum of testing times of three CLBs, 1688.

• Testing Resource Allocation

CLB	TPG	SAR	Test Scheme
CLB A	R1 (SARTPG)	R1	feedback
CLB B	R2 (SARTPG)	R2	feedback
CLB C	R2 (SARTPG)	R2	feedback

• **Hardware Overhead**

Component	Modification (Size)	Hardware Overhead (TRs)
R1	SAR (16)	318
R2	SAR (16)	318
R3	SCAN (16)	64
R4	SCAN (16)	64
Total		764

• **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB A	> 528
Session 2	CLB B	> 336
Session 3	CLB C	> 824
Total		> 1688

D.5.4 Structure 4

Structure 4 is generated from Structure 1 by eliminating R2 in order to reduce hardware overhead. The total hardware overhead is reduced from 846 TRs to 820 TRs. Testing time is increased from 1160 to 1688.

- **Testing Resource Allocation**

CLB	TPG	SAR	Test Scheme
CLB A	R1	R3	Pseudorandom
CLB B	R3	R1	Pseudorandom
CLB C	R4	R1	Pseudorandom

- **Hardware Overhead**

Component	Modification (Size)	Hardware Overhead (TRs)
R1	BILBO (16)	328
R2	SCAN (16)	64
R3	BILBO (16)	328
R4	TPG (16)	100
Total		820

- **Test Session Arrangement**

	Tested CLB	Testing Time
Session 1	CLB A	528
Session 2	CLB B	336
Session 3	CLB C	824
Total		1688

Appendix E

Derivation of $P_m(k)$ in 8.2.2

$P_m(k) = P_{m-1}(k-1)\lambda_m(k) + P_{m-1}(k)(1-\lambda_m(k+1))$, where $2 \le k < m$ and $2 \le m$.

$P_m(1) = P_{m-1}(1)(1-\lambda_m(2))$ for $k = 1$ and $2 \le m$.

$P_m(m) = P_{m-1}(m-1)(\lambda_m(m))$ for $k = m$ and $2 \le m$.

Substituting $P_m(k)$ to the expected value $E[D_m] = \sum_{k=1}^{m} kP_m(k)$ gives

$$
\begin{aligned}
E[D_m] &= P_{m-1}(1)(1-\lambda_m(2)) + mP_{m-1}(m-1)\lambda_m(m) \\
&\quad + \sum_{k=2}^{m-1} k[P_{m-1}(k-1)\lambda_m(k) + P_{m-1}(k)(1-\lambda_m(k+1))] \\
&= P_{m-1}(1) + \sum_{k=2}^{m-1} kP_{m-1}(k) \\
&\quad + \sum_{k=2}^{m-1} kP_{m-1}(k-1)\lambda_m(k) - \sum_{k=2}^{m-1} kP_{m-1}(k)\lambda_m(k+1) \\
&\quad - P_{m-1}\lambda_m(2) + mP_{m-1}(m-1)\lambda_m(m).
\end{aligned}
$$

Using the definition $E[D_{m-1}] = \sum_{k=1}^{m-1} kP_{m-1}(k)$

$$
\begin{aligned}
E[D_m] &= E[D_{m-1}] - P_{m-1}\lambda_m(2) + P_{m-1}(m-1)\lambda_m(m) \\
&\quad + \sum_{k=2}^{m-1} kP_{m-1}(k-1)\lambda_m(k) - \sum_{k=3}^{m-1} (k-1)P_{m-1}(k-1)\lambda_m(k) \\
&= E[D_{m-1}] + P_{m-1}(1)\lambda_m(2) + P_{m-1}(m-1)\lambda_m(m) \\
&\quad + \sum_{k=3}^{m-1} P_{m-1}(k-1)\lambda_m(k) \\
&= E[D_{m-1}] + \sum_{k=1}^{m-1} P_{m-1}(k)\lambda_m(k+1).
\end{aligned}
$$

The vita has been removed from the scanned document