Techniques for Seed Computation and Testability Enhancement for Logic Built-In Self Test

Dhrumeel V. Bakshi

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

Master of Science
in
Computer Engineering

Michael S. Hsiao, Chair
Patrick R. Schaumont
Sandeep K. Shukla

August 27, 2012
Blacksburg, Virginia

Keywords: LFSR Reseeding, Logic Built-in Self Test (LBIST), Satisfiability Modulo Theories (SMT), Integer Linear Programming (ILP), Test-point Insertion

Techniques for Seed Computation and Testability Enhancement for Logic Built-In Self Test

Dhrumeel V. Bakshi

ABSTRACT

With the increase of device complexity and test-data volume required to guarantee adequate defect coverage, external testing is becoming increasingly difficult and expensive. Logic Built-in Self Test (LBIST) is a viable alternative test strategy as it helps reduce dependence on an elaborate external test equipment, enables the application of a large number of random tests, and allows for at-speed testing. The main problem with LBIST is suboptimal fault coverage achievable with random vectors. LFSR reseeding is used to increase the coverage. However, to achieve satisfactory coverage, one often needs a large number of seeds. Computing a small number of seeds for LBIST reseeding still remains a tremendous challenge, since the vectors needed to detect all faults may be spread across the huge LFSR vector space. In this work, we propose new methods to enable the computation of a small number of LFSR seeds to cover all stuck-at faults as a first-order satisfiability problem involving extended theories. We present a technique based on SMT (Satisfiability Modulo Theories) with the theory of bit-vectors to combine the tasks of test-generation and seed computation. We describe a seed reduction flow which is based on the 'chaining' of faults instead of pre-computed vectors. We experimentally demonstrate that our method can produce very small sets of seeds for complete stuck-at fault coverage. Additionally, we present methods for inserting test-points to enhance the testability of a circuit in such a way as to allow even further reduction in the number of seeds.

To my family

Acknowledgments

Above everyone else, I would like to thank my graduate research advisor Dr. Michael Hsiao for his invaluable guidance and support for the entire duration of my graduate study. Through countless interactions with him in his office, group meetings, lectures and lunches I have learned what it means to be a good student, a good engineer and a good researcher.

I would like to express my sincere appreciation to Dr. Patrick Schaumont and Dr. Sandeep Shukla for serving on my thesis committee, for their suggestions and guidance through coursework and otherwise. I would like to thank SRC for funding and mentoring the work that was done towards this thesis.

I am grateful to Vijay Gangaram for giving me the opportunity to intern in his team at Intel Corporation. I have learned a lot about research in the real world and industry, thanks to his taking a keen interest in my work and providing valuable feedback. I am also thankful to Pallav Gupta for the numerous brainstorming sessions and code-reviews, which have helped me develop better coding habits. These skills have undoubtedly affected the quality of this work.

I am sincerely thankful to my high school math teacher Dhiren Patel, whose inspiration continues to motivate me to think logically and coherently.

I thank Gautham Chavali, Michael Steiner and Saparya Krishnamoorthy for their patience during my long rants about research when things did not work so well, and for their constant support and useful feedback. I am thankful to my friends at PROACTIVE - Supratik Misra,

Sarvesh Prabhu, Neha Goel, Min Li, Huy Nguyen, Chinmay Limaye, Mainak Banga, Nikhil Rahagude and Maheshwar Chandrasekar for fun times both inside and outside the lab.

I thank my friends Riddhi Malhotra, Jigar Dalsania, Tanay Shah, Avni Palekar and Harshada Sant for reinforcing my confidence whenever I need it.

Finally, I would like to thank my family for their encouragement and unconditional support in everything I do.

Dhrumeel Bakshi

Blacksburg

August 17, 2012

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Digital systems have found their way into virtually all aspects of our lives. Demands on the quality, efficiency and functionality of VLSI circuits are constantly rising. To satisfy these demands, more logic continues to be packed onto a single die. Aggressive logic synthesis and optimization techniques coupled with the employment of novel fabrication processes cause the likelihood of manufacturing defects to go up. Rigorous testing of manufactured chips is needed to ensure reliability of the shipped products. Testing also helps in repair and yield improvement by effectively diagnosing and analyzing the root cause of defects.

IC testing today is done largely by the use of external testers called Automated Testing Equipment (ATEs). Modern ATEs are extremely expensive and complex. The International Technology Roadmap for Semiconductors (ITRS) has estimated an exponential increase in the amount of test data that will be required to achieve satisfactory fault coverage. Figure 1.1 shows the projected values for number of test-patterns and test data requirements for single-stuck faults in the coming years [1]. ATEs may also operate at a much lower frequency than the operational frequency of the circuit. This leads to great increase in test cost due to longer test times and the expense of storing large amounts of test data in tester memory. External test cost is currently estimated to be as high as 25-30% of the total manufacturing cost [2]. While silicon cost is decreasing, test cost is continuously increasing [3].

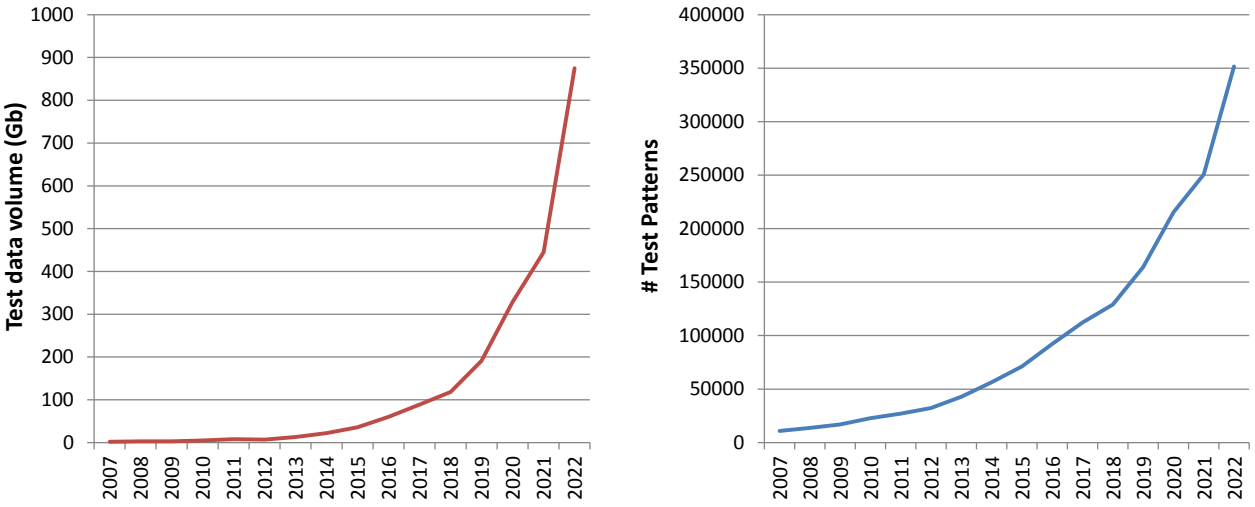


Figure 1.1: ITRS projections of test data requirements for stuck-at faults [1] (Used under fair use, 2012)

Design for Testability (DFT) has emerged as a popular area of research that aims to provide systematic techniques to reduce test cost by facilitating the ease of test development and test application. Built-in Self Test (BIST) refers to a specific DFT technique which enables a chip to test itself. As the size and complexity of functional logic being built into chips keep increasing, BIST methods aim to alleviate the cost of ATE-based testing. In BIST, generation of test patterns and analysis of output responses are both performed on-chip. This helps to either reduce ATE complexity or eliminate ATEs altogether. BIST allows for at-speed testing of the chip which is needed to test performance-related defects. Additionally, as test mechanisms are embedded into the chip, BIST has a better access to the core logic to be tested as compared to external testing. With BIST, a self-testable chip can test itself even after it is part of a system.

While BIST offers the aforementioned advantages, there are challenges that come with it. Inclusion of both Test Pattern Generator (TPG) and Output Response Analyzer (ORA) structures on chip increases the silicon area and associated hardware costs. To reduce the overhead costs, often very low hardware-overhead TPG structures capable of producing pseudo-random patterns are preferred. However, the patterns produced by these structures

are not optimal. The TPG structures used in BIST are Pseudo Random Pattern Generators (PRPGs), and they generally require a large number of vectors to be applied to the circuit in order to guarantee adequate defect coverage.

Most critically, achieving a high fault-coverage with BIST is a challenging problem for circuits containing random-pattern-resistant and/or hard-to-test faults. Various solutions have been proposed to tackle this problem. A popular technique known as *mixed-mode BIST* involves testing the circuit first using pseudo-random patterns followed by deterministic methods to cover faults missed by the random patterns. In this thesis, the focus is on a sub-class of mixed-mode BIST called *reseeding*, in which test-patterns are encoded as initial states or *seeds* of a TPG such as a Linear Feedback Shift Register (LFSR). A large amount of work has been done on computing seeds for LFSR-based pattern-generation. Most of previously proposed techniques attempt to compress pre-computed ATPG patterns by encoding them as seeds. In cases where the number of deterministic patterns to be encoded is large, these methods require many seeds to be stored on chip. Test-data compression based on LFSR reseeding continues to be an actively researched problem.

In this thesis, we propose methods to compute high quality LFSR seeds for a test-per-clock testing scheme. In our approach, we find seeds not to cover pre-determined patterns but to cover a subset of the faults. We try to get the most out of each seed, and we will experimentally demonstrate that our method can compute very small sets of seeds for complete stuck-at fault coverage. Additionally, we will describe methods to enhance the testability of the circuit via test-points and demonstrate that the enhanced circuit can be fully tested with even fewer number of LFSR seeds.

1.1 Contributions of the Thesis

We propose a new way to look at the problem of reseeding, wherein seeds are computed by considering target faults rather than target test patterns. We will show that this approach

can help us cover all faults in the circuit using very few number of seeds. The method does not require a prior step of test-pattern generation and works by effectively combining the test-generation and seed-computation processes. We make effective use of the capabilities of modern Satisfiability Modulo Theories (SMT) solvers and general purpose Integer Linear Program (ILP) solvers.

The specific contributions of the thesis are summarized below:

1. We cast the problem of test-generation and seed computation as a first-order satisfiability problem involving extended theories. We present a new SMT formulation for test-generation under LFSR constraints, which helps us compute seeds to 'chain' faults instead of pre-computed vectors. For solving this kind of a constraint-problem, we make effective use of SMT-solving using the theory of bit-vectors.

2. A complete flow for seed computation and reduction which uses the SMT model for fault-chaining. We propose a complete procedure for seed computation and reduction which iteratively arrives at a small set of high quality seeds. The method is based on identifying groups of faults to chain in every iteration. These groups are formed by finding cliques in a fault-independence graph.

3. Techniques for enhancing the testability of a circuit. Specifically, we propose methods to insert control and observation points with the objective of reducing the number of seeds to be stored for achieving complete stuck-at fault coverage. We also describe a method to implement control points through logic inversion, which allows us to reuse seeds to cover additional faults.

1.2 Thesis Outline

The remainder of the thesis is structured as follows:

- **Chapter 2** includes background and preliminaries relevant to this thesis. We discuss Design for Testability, Logic Built-in Self Test, Satisfiability Modulo Theories and Integer Linear Programming in brief. We also discuss related previous work.
- **Chapter 3** describes the SMT formulation to compute LFSR seeds to chain a set of faults.
- **Chapter 4** describes the complete flow for LFSR seed computation and reduction using SMT and ILP solving.
- **Chapter 5** describes methods for identifying and implementing test-points with the objective of reducing the number of seeds even further.
- **Chapter 6** concludes the thesis and outlines potential directions for future work.

Chapter 2

Background

This chapter provides the background and preliminaries relevant to the methods proposed in this thesis. In particular, we will discuss the following:

1. Design for Testability (DFT)
    - Scan-Design
    - Logic Built-In Self Test (LBIST)
    - Testability Analysis and Test-Point Insertion
2. Satisfiability Modulo Theories (SMT)
3. Integer Linear Programming (ILP)
4. Related Work relevant to this thesis
5. Motivation for this work

2.1 Design for Testability (DFT) - Overview

Owing to the size and complexity of modern VLSI circuits, design and test are no longer separate. Chips now need to be designed while paying attention to testing. *Design for testability (DFT)* refers to all those techniques and design-practices that enable us to adequately test the manufactured chips. DFT ensures that test development/execution time is kept low enough to be economical, while simultaneously simplifying the tasks of test, debug and diagnosis. DFT methods can be broadly classified into Ad-Hoc DFT and Structured DFT.

(a) **Ad-Hoc DFT**: This category consists of local modifications made to a circuit with the aim of enhancing its testability. These methods generally involve consultation of good design practices learned from experience. Some examples are making flip-flops (FFs) initializable, avoiding asynchronous logic and combinational loops, and inserting test points. Test point insertion may be supported by prior testability analysis. The ad-hoc approach needs manual inspection to identify potential testability issues and 'bad' design practices. Thus, results are unpredictable and it is difficult to automate in the general sense.

(b) **Structured DFT** includes techniques that involve the inclusion of special logic in a circuit with the intent of systematically enhancing the testability and allowing test-execution to be carried out in a well-defined manner. *Scan-design* is the most effective and widely used structured DFT technique. Another popular structured DFT methodology is *Logic Built-In Self Test (LBIST)*. We will discuss these techniques further in the following sections.

2.1.1 Scan-Design

Despite advances made in Automatic Test Pattern Generation (ATPG) for structural fault models (such as stuck-at faults, delay faults) test generation for sequential circuits remains

a very hard problem. The main reason for this is the need to justify hard-to-reach states using test patterns in order to test for certain faults. Certain faults may get 'propagated' to a state element, and thus require long patterns of state-checking to observe the fault effect.

Scan-design is a popular technique devised to obtain controllability and observability for the state-elements in the circuit. This is accomplished by turning the flip-flops into special *scan cells*. In the test mode, the FFs get configured into one or more shift-registers (called scan-chains), into which known values can be scanned-in from the outside world, and their previous values can be scanned out and observed. This direct control on the circuit state allows faults to be tested faster and more easily. There are three popular architectures that support scan-design.

(a) **Full-Scan:** In a full-scan design, all state-elements are turned into scan-cells. All inputs to the combinational logic can be controlled, so the circuit has Primary Inputs (PIs) and the state-elements which act as Pseudo-Primary Inputs (PPIs). Since scan-cell values are observable, the FFs also become Pseudo-Primary Outputs (PPOs). As a result, the circuit under test can be regarded as merely a combinational circuit, since all state elements are directly controllable and observable. Thus, combinational ATPG can now be used to generate tests for faults in the combinational logic. Multiple scan-chains may be used to reduce the serial scan-in/scan-out times.

(b) **Partial-Scan:** A partial-scan design is where only a subset of the FFs in the design are converted to scan-cells. This is to avoid scanning those flip-flops that the designers do not wish to be scanned, such as those on the critical paths. Depending on the configuration of partial-scan, either combinational or sequential ATPG may be required to generate test patterns. Considerable amount of work has been done on choosing the set of flip-flops that form scan-registers. For example, storage elements may be selected to remove sequential feedback loops [4,5]. This helps reduce test-generation complexity significantly, and helps achieve higher fault coverage.

(c) **Random-Access Scan:** The random-access scan technique makes each scan-cell uniquely

addressable, like cells in a Random Access Memory (RAM). This helps overcome the problems of high test-power due to switching and the difficulties associated with scanning in values serially. However, this architecture may be less preferred due to its large hardware overhead.

Scan-design is a very powerful DFT methodology, and it now acts as the backbone of more advanced DFT methods. Despite its overheads, full-scan design continues to be increasingly preferred in industry.

2.1.2 Logic Built-In Self Test (LBIST)

LBIST refers to a DFT methodology in which *test* is one of the functions of the system/chip. Certain additional constructs are built into the circuit in order to allow the circuit to test itself. A general view of LBIST structure is shown in Figure 2.1. The main components are a Test-Pattern Generator (TPG), an Output Response Analyzer (ORA) and a test control mechanism. The TPG may include a pattern-generator along with optional auxiliary read-only storage. The ORA typically includes a response-compactor along with a signature analyzer which signals any differences detected from a stored reference value. ORA often also contains diagnostic logic. Note that Figure 2.1 only shows a typical view. In practice, a TPG may generate patterns for one or more CUTs and there may exist separate TPG+ORA structures for different partitions of the circuit.

LBIST has become popular owing to the advances in powerful structured DFT techniques such as scan-design, test-data compression techniques and efficient output-response analysis techniques such as signature analysis and transition-count testing. The advantages of LBIST over conventional ATE-based testing are summarized below:

- As patten-generation and output-analysis are done on-chip, LBIST helps reduce ATE complexity and cost or eliminate ATEs completely.

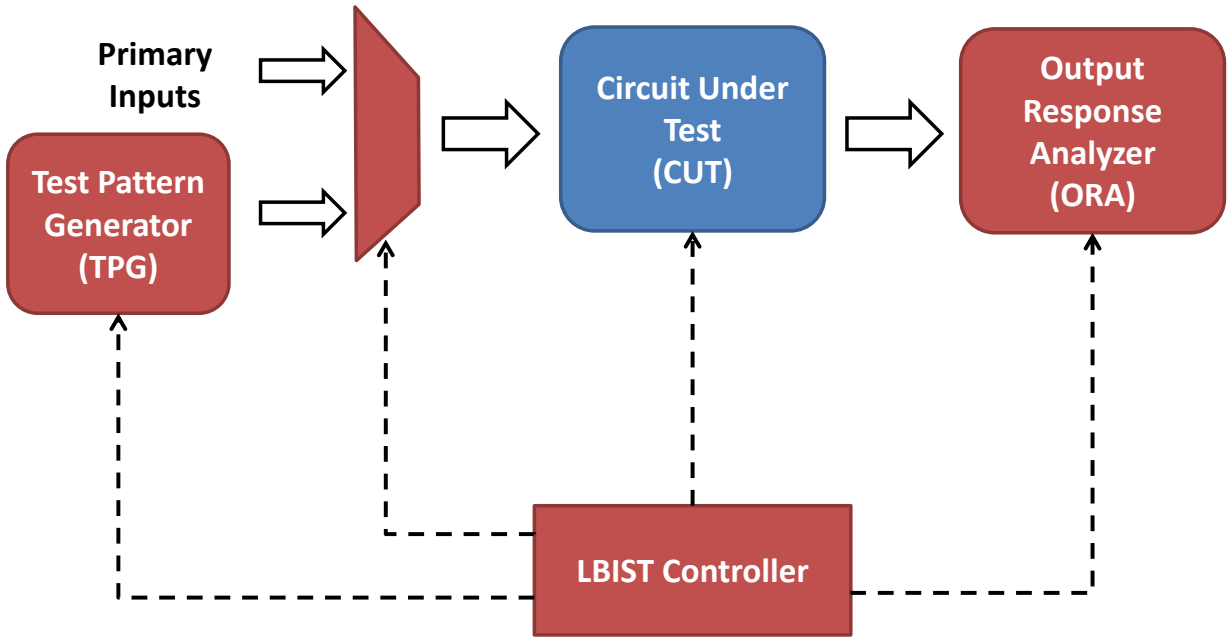


Figure 2.1: General structure of LBIST

- LBIST enables at-speed testing of circuits, which may not be possible with external ATEs. At-speed test is essential for detecting performance-related defects.
- As test mechanisms are embedded into the chip, LBIST has a better access to the core logic to be tested.
- A BIST-ed chip can be tested even after it is part of a system. It allows for the possibility of testing both **Online** (either concurrently or non-concurrently with functional operation), and **Offline** (to perform either functional or structural testing).
- LBIST simplifies test-partitioning and fault-diagnosis.
- It has been argued that while the cost of ATEs *increases* with each process generation, the associated BIST cost(overhead) *decreases* [6].

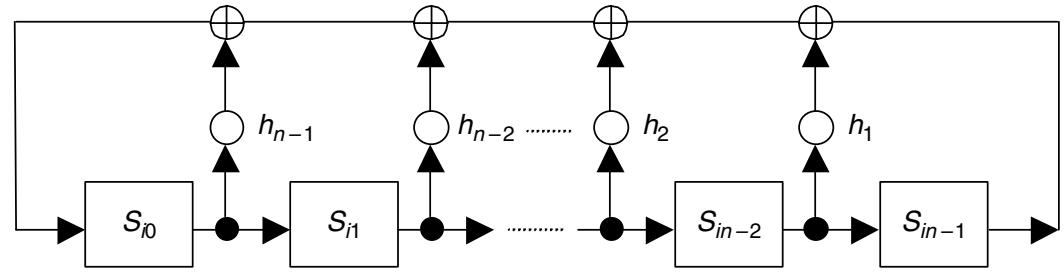
2.1.2.1 Test Pattern Generation for LBIST

Various pattern generation schemes have been studied for LBIST. **Exhaustive-testing** involves the application of all possible 2^n input patterns to an n-input combination circuit. This method is impractical even for a moderately large n. **Pseudo-exhaustive** testing is a scheme which aims to approximate exhaustive-testing by partitioning the circuit and then exhaustively testing each partition. Partitioning may be done by considering separate fan-in cones of influence for each primary output [7] or dividing the circuit into segments / subcircuits by methods such as described in [8]. **Pseudo-Random testing** is the most popular test-generation scheme. A Pseudo-Random Pattern generator (PRPG) [9] is used to generate patterns that satisfy most of the required properties of random numbers while being predictable. A large amount of work has been done on estimating required test-length and fault-coverages for pseudo-random patterns.

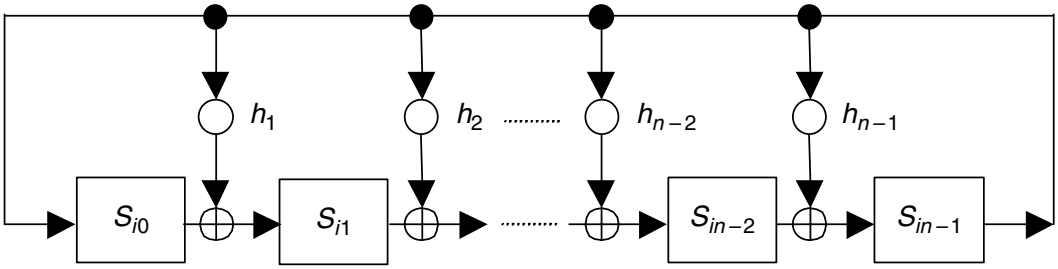
Various structures such as Cellular Automata, ROMs, binary-counters and their modifications have been studied for TPG. **Linear Feedback Shift-Registers** (LFSRs) are the most commonly used structures for PRPG. The main advantages of LFSRs are their low hardware overhead and output patterns with good quality of randomness. Figure 2.2 shows the two common types of LFSRs.

An **External-XOR/Standard/Fibonacci LFSR** is a circular shift-register with a feedback path composed of one or more XOR gates whose inputs are derived from specific bits of the register. The bits contributing to the feedback path are known as 'taps'. Thus, an external-XOR LFSR has a feedback bit which is a linear (modulo-2) sum of two or more of its bits. In an **Internal-XOR/Modular/Galois LFSR**, each XOR gate is placed between two adjacent LFSR stages (D flip-flops). The internal structure of each kind of LFSR in Figure 2.2 is described by a *characteristic polynomial* of degree n

$$f(x) = 1 + h_1 x + h_2 x^2 + \cdots + h_{n-1} x^{n-1} + x^n$$



(a) n-stage External-XOR/Standard LFSR

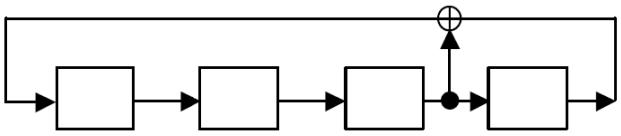


(b) n-stage Internal-XOR/Modular LFSR

Figure 2.2: LFSR types and general structure [10]

where each symbol h_i is either 1 or 0, indicating the presence or absence, respectively, of a tap at the i^{th} location. An LFSR can be seen as a finite state machine with period T whose value depends on the characteristic polynomial $f(x)$. An internal-XOR and an external-XOR LFSR of degree n having the same characteristic polynomial have distinct but isomorphic state diagrams. Hence, any mathematical analysis which applies to one kind of LFSR automatically applies to the other.

For certain characteristic polynomials (known as **primitive polynomials**), the LFSR produces a maximum-length sequence consisting of all 2^{n-1} possible non-zero states. Such an LFSR is called a **maximal-LFSR**. An example is shown in Figure 2.3. $f(x) = 1 + x + x^4$ is a primitive-polynomial, and the LFSR goes through all 15 non-zero states before repeating. The output sequence of a maximal-LFSR satisfies many empirical criteria for randomness [11]. This makes LFSRs a very attractive option for TPG since they can produce seemingly random, yet predictable output sequences with a very low hardware cost.



(a) 4-bit External-XOR LFSR

0	**0**	**0**	**0**	**1**	8	0	1	0	1
1	1	0	0	0	9	1	0	1	0
2	0	1	0	0	10	1	1	0	1
3	0	0	1	0	11	1	1	1	0
4	1	0	0	1	12	1	1	1	1
5	1	1	0	0	13	0	1	1	1
6	0	1	1	0	14	0	0	1	1
7	1	0	1	1	15	**0**	**0**	**0**	**1**

(b) State sequence

Figure 2.3: A 4-bit maximal-LFSR, $f(x) = 1 + x + x^4$

2.1.2.2 LBIST Fault-Coverage Enhancement

LFSR-patterns are far from optimal for testing modern VLSI circuits. In a circuit, there may exist many *random-pattern resistant* faults [12]. A PRPG scheme thus needs very long run times to reach a satisfactory fault coverage. Some solutions that have been proposed to address this problem are summarized below:

(a) **Test-point insertion** involves modification of the circuit under test by improving the detection probability of certain faults. This method will be discussed further in Section 2.1.3.

(b) **Hybrid-BIST** is applicable to manufacturing testing where an external tester is used. Additional test-content is loaded from tester-storage with the intent of covering faults not detected by pseudo-random testing. Hybrid-BIST methods are not applicable for in-the-field testing.

(c) The use of **weighted-random pattern generation** has been proposed to increase the probabilities of covering random-pattern resistant faults [13–15]. These methods generally require a large amount of hardware overhead to implement, mainly because of the need to store and implement multiple sets of weights.

(d) **Mixed-mode BIST** refers to a class of methods which aim to cover faults missed by PR

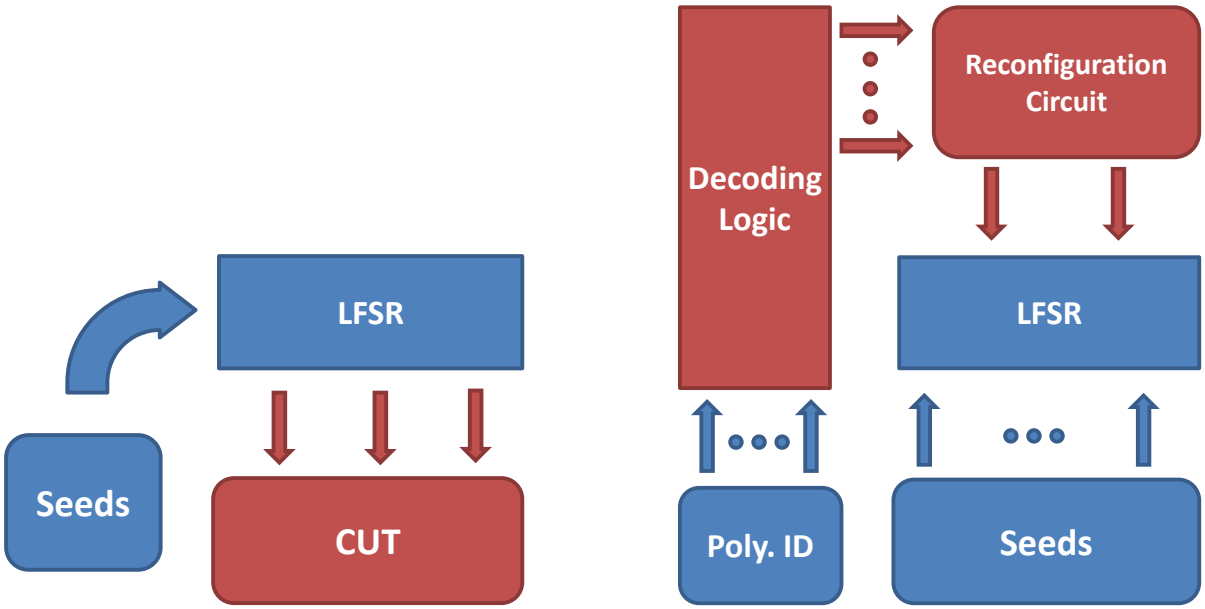
patterns, by using on-chip hardware to generate deterministic patterns. Several schemes have been proposed for designing a *special-purpose counter* to generate deterministic test-patterns (e.g. [16, 17]). However, binary counters generally entail a higher hardware overhead compared to LFSRs. Other schemes involve *ROM-based* pattern storage including compression techniques to reduce the ROM-size (e.g. [18, 19]). A large number of methods have been proposed for performing deterministic testing with LFSRs. Some of these methods rely on *mapping logic* such as bit-fixing, bit-flipping etc. to transform the LFSR patterns to deterministic ATPG patterns.

LFSR-reseeding is a class of mixed-mode BIST in which test-patterns are encoded as initial-configurations of an LFSR, called 'seeds'. Since seeds require less storage as compared to the test-patterns themselves, a set of seeds may be stored on an on-chip ROM. Special mechanism is built to allow loading the LFSR with a stored seed, and then running it in autonomous mode for a pre-determined number of test cycles. Other methods in reseeding have also proposed using a *reconfigurable* LFSR capable of changing not only its initial configuration, but also its characteristic polynomial. Polynomial and seed pairs are computed and stored on-chip to realize a Multiple-Polynomial LFSR (MP-LFSR). Both kinds of LFSR reseeding schemes are shown in Figure 2.4(b). A discussion of relevant work on LFSR reseeding will follow in Section 2.4.

2.1.2.3 LBIST Architectures

Owing to the versatility of LBIST, many different LBIST architectures have been proposed. We will only discuss a relevant representative sample here. The reader is directed to [10, 20, 21] for a more descriptive treatment.

Test-per Scan: In a test-per scan scheme, the PRPG is run for a number of cycles during which its output bits are serially shifted into one or more scan-chains of the CUT. At the same time, values of cells in the scan-chains are shifted out to a signature analyzer such as a



(a) Single-Polynomial LFSR (b) Multiple-Polynomial LFSR [10]

Figure 2.4: LFSR reseeding schemes

Single-Input Signature Register (SISR) for a single scan-chain or a Multiple-Input Signature Register (MISR) for multiple scan chains. The number of shift cycles is dictated by the length of the longest scan-chain in the design. An example of a test-per scan architecture is the *Self Testing Using MISR and Parallel SRSG (STUMPS)* architecture [22] which has been widely used in industry. Figure 2.5 shows a typical layout of STUMPS. The optional phase-shifter and phase-compactor modules help reduce the length of the PRPG and MISR, as well as improve the randomness of the PRPG output.

Test-per Clock: In this scheme, a test-pattern is applied to the CUT at every clock cycle. It results in a reduced test length as compared to test-per scan, but a larger PRPG/MISR may be required. Figure 2.6(a) shows a typical test-per clock scheme, where the outputs of the PRPG at every clock cycle are used to test the CUT. For a CUT with a large number of inputs, a combination of LFSR+SR may be used as shown in Figure 2.6(b). Advanced register-reconfiguration methods like Built-In Logic Block Observer (BILBO) [23] and modifications

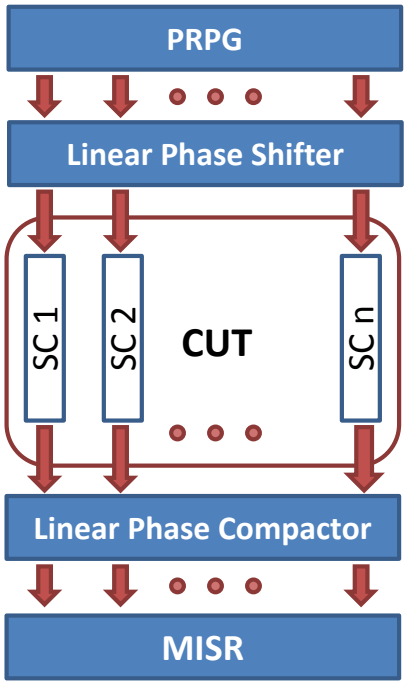


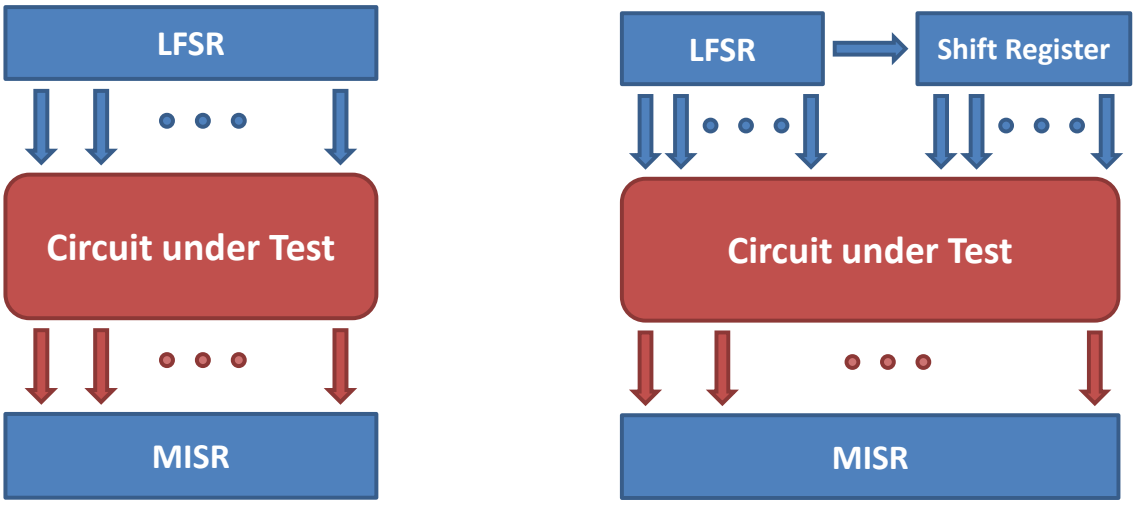
Figure 2.5: STUMPS architecture [10]

such as Concurrent-BILBO (CBILBO) [24] have been proposed. These techniques allow the circuit state elements to themselves be used for pattern-generation and response analysis in different modes or phases.

2.1.3 Testability Analysis and Test-Point Insertion

Test-point insertion is a common method of modifying the circuit so that it becomes easier to test. **Testability** is defined in [10] as "The process of assessing the testability of a logic circuit by calculating a set of numerical measures for each signal in the circuit." These numerical measures are often termed *Testability-metrics*. Testability analysis is useful for identification of potential testability issues at certain signals as well as for guiding ATPG decision-making.

The two most popular testability metrics are:



(a) Test-per clock

(b) Test-per Clock for Large Input CUT

Figure 2.6: Typical Test-per-clock schemes [20]

(a) **Sandia Controllability/Observability Analysis Program (SCOAP)** [25] is a topology-based testability metric. It assigns a controllability and an observability value to each signal, which reflects the difficulty of setting a value to or observing the value of the signal, respectively. Essentially, SCOAP aims to identify the approximate number of primary-inputs that must be specified, in order to control or observe a given signal. SCOAP is computationally efficient, but tends to produce inaccurate results for circuits with large number of *reconvergent-fanouts*.

(b) **Controllability Observability Program (COP)** [26] is a probability based testability metric. It estimates the probabilities $C0$ and $C1$ of controlling signal s to a 0 or a 1 value from the primary inputs. The primary-inputs are typically assigned a probability (controllability) value of 0.5. COP also assigns an observability value $O(s)$, which estimates the probability of being able to observe the value of signal s at a primary output. COP is computationally efficient and known to be very effective in analyzing the random testability of the circuit, in spite of the approximating assumption that the signal probabilities at different inputs of a logic gate are mutually independent.

COP testability computation rules are summarized in Table 2.1. First, the controllabilities are computed for all signals in topological order from PIs to POs (assigning PI-controllability value to 0.5). Next the observabilities are computed in reverse-topological order, starting with and assigning an $O(s)$ of 1 to the POs. The computed testability values may then be used to identify potential sites for hard-to-test faults, and test-points are inserted.

A **control-point** is inserted at a signal for which the controllability value was computed to be close to 0 or 1. A control point typically includes a MUX or AND-OR gates to allow value-injection in the test-mode. The point may be driven by a primary input, an existing or dedicated scan-cell output, another signal within the circuit or by a constant-value. Control-point activation has been an area of study. Various methods to activate control-points have been proposed such as concurrent activation of all control-points in the test mode, activation using pattern-decoding logic, multi-phase activation etc.

An **observation-point** is inserted at hard-to-observe nodes in the circuit, with the aim of being able to detect all faults that 'propagate' to that node. An observation-point may be connected to a primary output or existing scan-cell through a MUX, or to a dedicated scan-cell. Schemes which make use of an XOR-gate tree to multiplex multiple observation-points to a single output have also been proposed.

2.2 Satisfiability Modulo Theories (SMT)

SMT is a decision problem related to the evaluation of logical formulas with respect to some background theories. It can be viewed as a generalization of the Boolean satisfiability problem (SAT), which is the problem of determining if there exists a valuation to the Boolean variables in a propositional formula so as to make the formula evaluate to *true*. In an SMT instance, some of the binary variables are replaced by predicates or 'clauses' in some decidable first order theory $\mathcal{T}$ or a combination of theories.

Propositional satisfiability solving has seen many advances in the last couple of decades

Table 2.1: COP computation rules [10] (Used under fair use, 2012)

(a) Controllability rules

	0-Controllability (Primary Input, Output, Branch)	**1-Controllability (Primary Input, Output, Branch)**
Primary Input	p_0	$p_1 = 1 - p_0$
AND	1 – (output 1-controllability)	$\prod$ (input 1-controllabilities)
OR	$\prod$ (input 0-controllabilities)	1 – (output 0-controllability)
NOT	Input 1-controllability	Input 0-controllability
NAND	$\prod$ (input 1-controllabilities)	1 – (output 0-controllability)
NOR	1 – (output 1-controllability)	$\prod$ (input 0-controllabilities)
BUFFER	Input 0-controllability	Input 1-controllability
XOR	1 – 1-controllability	$\sum (C1(a) \times C0(b), C0(a) \times C1(b))$
XNOR	1 – 1-controllability	$\sum (C0(a) \times C0(b), C1(a) \times C1(b))$
Branch	Stem 0-controllability	Stem 1-controllability

(b) Observability rules

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

(eg. Chaff [27], GRASP [28]). Most modern SAT solvers are based on the popular DPLL procedure [29, 30]. Even though SAT is the classic NP-complete problem, present day SAT solvers are extremely efficient for most applications. SMT has recently gained much popularity due to its expressive power and the need for reasoning at higher levels of abstraction. SMT provides a richer modeling language as compared to SAT. Many complex decision problems are easily mechanically translated from the problem domain to the language of a theory supported by SMT. In contrast, in order to effectively utilize a SAT solver additional work must be done to convert the problem to the propositional domain (typically a CNF formula), and ensure correctness of the model.

SMT solving is currently an active area of research. It requires capabilities for heavy Boolean reasoning combined with an ability to efficiently reason in decidable first order theories of interest. Most modern SMT solvers are based on the so called *Lazy* SMT solving approach [31], which attempts to combine a SAT solver with theory specific decision procedures (aka. theory solvers). Since the SAT solver is generally based on the DPLL procedure, lazy SMT is also sometimes referred to as DPLL($\mathcal{T}$). The SAT solver is used to truth assignments to satisfy the Boolean abstraction of the SMT model, while the theory solver can be called on demand to check the consistency of theory-specific literals corresponding to each assignment. Advances continue to be made to make lazy SMT more and more efficient by heuristics to capitalize on the advantages of integrating a SAT solver with a theory solver. In contrast to lazy SMT solving, *Eager* SMT solving attempts to convert an SMT model into an equisatisfiable Boolean formula which can be then passed to a SAT solver. This approach frequently suffers from the problem of exponential blow-up in formula size and inefficiency due to loss of information about the correlation among variables.

Some of the popular SMT solvers of today are Z3 [32], Yices [33] and MATHSAT [34]. An effort worthy of mention is the SMT-LIB initiative [35], which aims to provide standardized definitions for background theories, a standard input-output language and a library of benchmarks for SMT solvers. Most SMT solvers of today are compatible with the SMT-LIB language.

Examples of background theories supported by SMT solvers are Equality and Uninterpreted Functions ($\mathcal{EUF}$), Linear Arithmetic over the reals ($\mathcal{LA}(\mathcal{Q})$) and integers ($\mathcal{LA}(\mathcal{Z})$), Difference Logic ($\mathcal{DL}$), Arrays ($\mathcal{AR}$) and the theory of bit-vectors ($\mathcal{BV}$) and their combinations. SMT solvers find applications in various domains including formal verification, automated test-generation for both hardware and software, compiler optimization and verification of RTL circuits and microcode [31].

2.3 Integer Linear Programming (ILP)

Linear Programming (LP) is a mathematical method to determine the 'best' solution for a decision problem. LP belongs to the sub-class of mathematical optimization problems where the objective (cost) function as well as the constraints are expressed using linear functions over the decision variables. ILP is a special case of LP where the variables are constrained to be integer valued. If only some of the variables are integers, it is referred to as Mixed Integer Programming (MIP). LP problems consist of an objective function to optimize, a set of constraints (equalities or inequalities) that the solution must satisfy and variable bounds (constraints on the range of values variables may take).

While LP solving has polynomial complexity, MIP and ILP are known to be NP-Hard. Various algorithms for solving LP and ILP problems have been proposed. Popular methods include the Simplex algorithm [36] and Interior Point methods [37]. Popular methods for solving ILP include Branch and Bound, Cutting Plane methods and their variants supported by many different heuristics. Very efficient tools are available for solving practical LP and ILP problems. Gurobi [38] and IBM ILOG CPLEX [39] are examples of commercially available optimizers that can efficiently handle problems with thousands of variables.

LP and ILP solving find applications in various area of business, economics and engineering. In the field of electronic design automation, ILP based methods have been used for power optimization, FPGA placement and routing, generation of minimum size prime implicants in

logic synthesis [40] etc. In the area of testing of digital systems, ILP has been previously used for test compaction and compression [41–43] as well as for various optimization problems in test generation [44–49].

2.4 Related Work

The problem of deterministic testing using LFSRs has been widely studied. Seed computation for deterministic testing was first described in [50]. In [51, 52], techniques have been proposed to encode deterministic test-patterns as seeds of an LFSR whose configuration (polynomial) can be altered during test. Techniques for enhancing test-generation for reseeding schemes have been proposed in [53, 54]. Other methods for reduction of the number of required seeds rely on identification and reduction of seeds by simulation and seed ordering/encoding methods (e.g. [6, 55]). A large number of methods such as [56] describe techniques for efficient hardware encoding of the seeds. Several techniques have been proposed for altering the patterns produced by LFSRs so that pre-determined patterns may be realized ([57–59]). Another area of work is identifying LFSR seeds and polynomials using genetic algorithms ([60, 61]).

Most of the previous methods for seed computation attempt to compute seeds for pre-computed test patterns. This involves grouping and ordering test-patterns followed by solving of linear-equations to get a seed for each group. Various problems with this approach are identified in the next section. Our method removes these problems by computing seeds for faults instead of pre-computed vectors. The method described in [62] works by identifying the exact location of test-patterns in the LFSR state-cycle by computation of Discrete Logarithms. However, these logarithms need to be computed for all possible vectors for target faults, which is impractical for modern designs.

Work has been done recently to combine the processes of test-generation and seed-computation using SMT solving [63]. As will be discussed in Section 3.3, the method in [63] is time-

expensive due to large SMT-formula size and search space. The method also has problems of fault-masking due to multiple injection in each frame. In this work, we propose a hybrid method based on simulation+analysis which eliminates these problems and makes the problem tractable for larger circuits as well.

Many methods have been proposed targeted towards the identification as well as implementation of test-points to improve circuit testability. We will briefly mention here some of these methods relevant to our work. The test-point problem was first proposed in [64]. Optimal test-point insertion was shown to be NP-complete for circuits with reconvergent fanout in [65]. Methods have been proposed which rely on fault-simulation to identify test-point candidate locations [66,67]. Methods have also been proposed which make use of testability metrics such as COP [26] to guide test-point selection. In [68], a method is proposed to select test-points by identifying sectors of hard-to-detect faults in the circuit. [69] described a method of test-point selection based on computation of the 'gradient' of a COP-based cost function for a given test-point candidate site. The method of [69] was enhanced in [70], where a hybrid method is used which relies both on gradients and explicit testability-computation. Probabilistic fault-simulation was used in [71] to guide test-point selection. Also, the test points were activated in multiple-phases to share value-drivers and reduce overhead. A path-tracing based method was used in [72] to identify gates which block the excitation and propagation of faults, as potential test-point insertion sites. The method in [73] aimed to identify test-points in a somewhat similar way, by recognizing the root-causes of poor testability. In this work we propose a method of test-point selection that can help us achieve BIST seed reduction.

2.5 Motivation

There are many advantages to LBIST as described in Section 2.1.2. The main issue with LBIST is achieving a satisfactory fault coverage with a feasible number of test patterns to be applied to the Circuit Under Test (CUT). Pattern generators such as Cellular Automata

(CA) and LFSRs can help produce very high quality random patterns with a very low hardware overhead. For modern designs which generally include a significant number of random-pattern-resistant faults, pseudo-random tests are no longer sufficient to test the circuit adequately.

An n-bit LFSR with a primitive characteristic polynomial is capable of cycling through $2^n - 1$ unique patterns before repeating. It becomes infeasible to utilize all of these test patterns beyond an LFSR length of around 30. Reseeding is an effective technique to increase the fault-coverage of the test patterns produced by an LFSR. As described in Section 2.1.2, LFSR seeds are stored on-chip for BIST. Each seed is then used to produce a number of test patterns. A large number of methods have been proposed to utilize the output bit-stream of an LFSR to populate the scan-chain (or multiple scan-chains) with pre-determined test-patterns determined by an ATPG tool. In a test-per-scan scheme, each seed acts as a *compressed* version of a number of test-cubes. On running the LFSR, the seed 'expands' into the required patterns. In LFSR-based compression schemes each seed can produce a small number of test-cubes (generally between 5 and 10). If the number of ATPG patterns to cover all random-pattern-resistant faults is large, a large number of seeds will need to be stored.

In this thesis, we present techniques to compute seeds for a test-per-clock scheme. Here, the contents of the LFSR in each cycle constitute a test vector to be applied to the CUT. A seed is the initial state of the LFSR from which it will run through good-quality vectors. The test-sequence generated by a single seed is illustrated in Figure 2.7. Since the number of seeds to be stored is to be kept as low as possible, an extremely tiny fraction of the LFSR state-cycle is utilized for testing. Good quality patterns may be spread all across the LFSR cycle. Rather than scattering the seeds around the vector-space, we would like to identify a small set of high-quality seeds which are capable of producing patterns to target the random-pattern-resistant faults.

The bit-stream/patterns produced by an LFSR are seemingly chaotic, yet predictable. Even though vectors that an LFSR will produce starting from a given seed can be exactly determined,

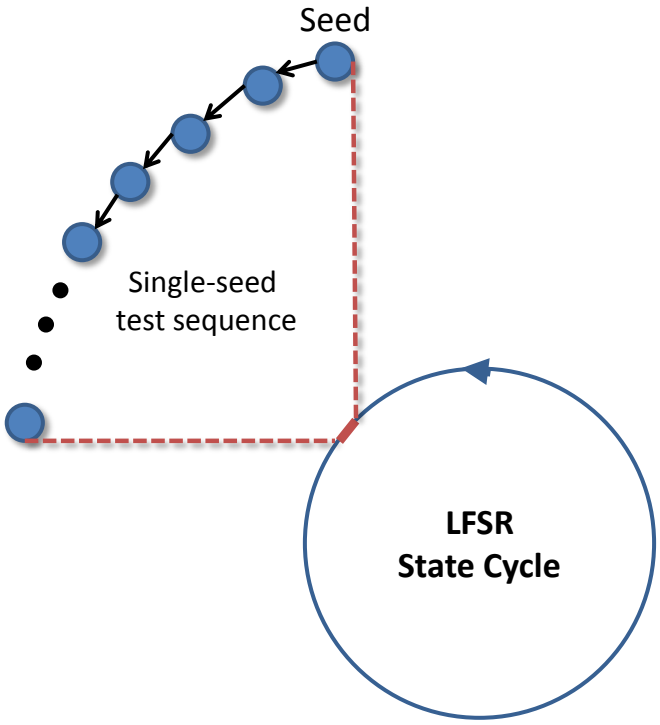


Figure 2.7: LFSR single-seed test sequence

it is difficult to reason about long LFSR sequences at a time due to its pseudo-random nature. The goal of this work is to enable the use of an LFSR to perform deterministic testing. We propose a combined methodology for seed-computation and test-generation in order to reduce/remove our dependence on pre-generated deterministic tests. We will describe techniques for computing a very small set of LFSR seeds using the capabilities of present day SMT and ILP solvers. We also propose a method to identify test-points for enhancing the testability of the circuit with the goal of reducing the set of seeds even further.

Chapter 3

Seed Computation Using SMT

In this chapter, we describe the SMT formulation for test-generation under LFSR-constraints, which we use for *fault-chaining*. The vectors generated form the seeds with which multiple faults will be detected within the LBIST environment.

As described in Section 2.1.2, LFSRs can be used in various configurations for pattern generation and application. In this thesis, we have used the *Fibonacci LFSR* configuration (aka. *external* LFSR). We assume a test-per-clock scheme as was shown in Figure 2.6(a), where the contents of the LFSR at every cycle constitute exactly one test-vector to be applied to the circuit under test. We have used an LFSR of length equal to the number of primary inputs of the combinational circuit to be tested. Circuits with sequential elements need to be converted to their corresponding *full-scan* version before applying our method. However, the method is applicable to other LFSR configurations as well (e.g., Internal-XOR or Galois-type LFSR). Also, our method can be applied to other BIST configurations involving modified architectures of the scan-chain layout. This can be done by making necessary changes to the SMT formulation discussed in the next section.

3.1 Key Idea

Most of the previous approaches to the computation of LBIST seeds attempt to chain a set of pre-determined test vectors. There are inherent problems with such approaches. Before going into the problems, we will first define some related terms. We use the term *LFSR* to represent an LFSR configuration along with its polynomial.

L-Distance Given a test set, T, comprising K bit-vectors $\{T_1, \ldots, T_K\}$ (the vectors may contain don't-care bits), and an LFSR, we define the *L-Distance* of the set T with respect to the given LFSR to be no greater than an integer L if and only if there exists a *seed*, S, which can generate all the vectors of T in any order, when the LFSR is run for L number of cycles.

F-Distance Given a set of K faults, F, $\{F_1, \ldots, F_K\}$, and an LFSR, we define the *F-Distance* of the set F with respect to the LFSR to be no greater than an integer L if and only if there exists a *seed*, S, such that the LFSR when run for L cycles starting from S, produces a set of test-vectors sufficient to detect all faults in F, in any order.

Chainability A set of bit-vectors (faults) is defined to be *chainable* within L cycles of a given LFSR if and only if the *L-Distance (F-Distance)* of the set is less than or equal to L.

Please note that the term *F-Distance* implicitly assumes that a specific fault-model has been selected for consideration. In this thesis, the *single-stuck* fault model has been assumed. Although we use a test-per-clock scheme, the terms may be easily generalized for other BIST architectures such as LFSR-SR (Figure 2.6(b)).

Given a stuck-fault in a circuit, there usually exist a large number of vectors which can detect the fault. For a set of hard-to-chain faults, the choice of the exact vectors to use for chaining these faults has a direct impact on the number of seeds needed. Realizing that LFSR vectors are chaotic (but predictable from an initial seed), we can see that even making slight changes

in the test-vector bits has the potential to alleviate or magnify the problem of *vector-chaining*. Thus, there may exist alternatives to vector-chaining and give smaller set of seeds.

We propose a method of seed-computation which tries to 'chain' faults rather than pre-determined test-patterns. We believe this to be a more effective way to view the problem of seed computation. In this work, we use an SMT formulation for *fault-chaining* and integrate it into a broader cost-effective method for computing LFSR seeds. This method allows us to select a small set of very high quality seeds. In effect, by considering the chainability of faults (*F-distance*) instead of chainability of vectors (*L-distance*), we offer a tighter integration of the test-generation process with seed computation. In addition, this method shows potential in overcoming the problems associated with vector-chaining and gives good quality solutions with small computational cost.

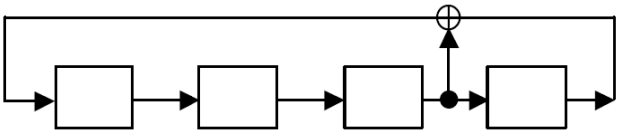
3.2 SMT Formulation

We now describe the SMT formulation for test generation under LFSR constraints. Given the netlist of the circuit under test and a set of single-stuck-faults to *chain*, we build an SMT model. Running an SMT solver on this model can help us find a seed that can chain and estimate the *F-Distance* of the set of faults. As described in Section 2.2, today's SMT solvers are capable of handling many first-order theories. In this work, we model the problem by combining the domains of plain Boolean (propositional) logic and quantifier-free bit-vector theory ($\mathcal{QF}$-$\mathcal{BV}$).

Suppose we have the gate-level netlist of a circuit, and a set of K single-stuck-faults $F = \{F_1, \ldots, F_K\}$. Also given to us is the LFSR-structure, and a bound L which is a heuristic bound set on the F-Distance. The SMT formulation to determine whether the set F has an *F-Distance* $\leq L$, is a formula that consists of the following sub-formulas:

(a) **LFSR constraints C in the theory of bit-vectors**

These constraints consist of L bit-vector variables $S_0, \ldots, S_{L-1}$. S_0 represents the *seed*, while the rest of the variables represent the contents (*state*) of the LFSR that can be derived in successive cycles. We know that an external-LFSR can be seen as a circular shift-register that shifts right in every cycle, with the leftmost bit defined by the feedback XOR-network. Thus, every bit-vector variable S_i is defined in terms of S_{i-1} using a bit-vector right-shift operation. Additionally, the leftmost bit of S_i is a function of some of the individual bits of S_{i-1}. These constraints are easily expressible in SMT, since $\mathcal{QF}$-$\mathcal{BV}$ allows for *extraction* of certain bits from bit-vectors, and specifying Boolean propositions using them. The initial seed S_0 is an unspecified variable, and each one of the S_i variables can potentially detect one or more of the faults in set F.



$\mathbf{S_i}$	d	c	b	a
$\mathbf{S_{i+1}}$	$a \oplus b$	d	c	b
$\mathbf{S_{i+2}}$	$b \oplus c$	$a \oplus b$	d	c
$\mathbf{S_{i+3}}$	$c \oplus d$	$b \oplus c$	$a \oplus b$	d
$\mathbf{S_{i+4}}$	$a \oplus b \oplus d$	$c \oplus d$	$b \oplus c$	$a \oplus b$

Figure 3.1: LFSR shift example

Figure 3.1 shows an example of a 4-bit LFSR, along with its contents in four successive cycles starting with the bit-vector $S_i = dcba$. Equation 3.1 shows how the constraint for S_{i+1} is modeled in terms of S_i, for the LFSR shown in Figure 3.1. Here $\#$ represents the bit-vector concatenation operation and $\oplus$ represents the XOR operation.

$$S_{i+1}[3:0] = (S_i[1] \oplus S_i[0]) \ \# \ S_i[3:1] \tag{3.1}$$

(b) **Fault detection formula $\mathbf{D_i}$ for every fault F_i**

Each formula $\mathbf{D_i}$ comprises a formula $\mathbf{G_i}$ (*good circuit*) and a formula $\mathbf{B_i}$ (*faulty circuit*). The constraints in both $\mathbf{G_i}$ and $\mathbf{B_i}$ are modeled with Boolean propositions using bit-vectors of size 1 for every gate, from the gate-level structure of the circuit. $\mathbf{G_i}$ models

fault-free operation while $\mathbf{B_i}$ is the same formula with the node at the fault location being replaced by constant-value drivers. Both $\mathbf{G_i}$ and $\mathbf{B_i}$ have the same input vector T_i, representing a *test* for fault F_i. The outputs of the two circuits are then fed, pair-wise, via XOR gates, effectively forming a *miter circuit*. In this miter (of G_i and B_i), the output D_i is *true* if and only if the vector T_i causes at least one output of $\mathbf{G_i}$ and $\mathbf{B_i}$ to differ.

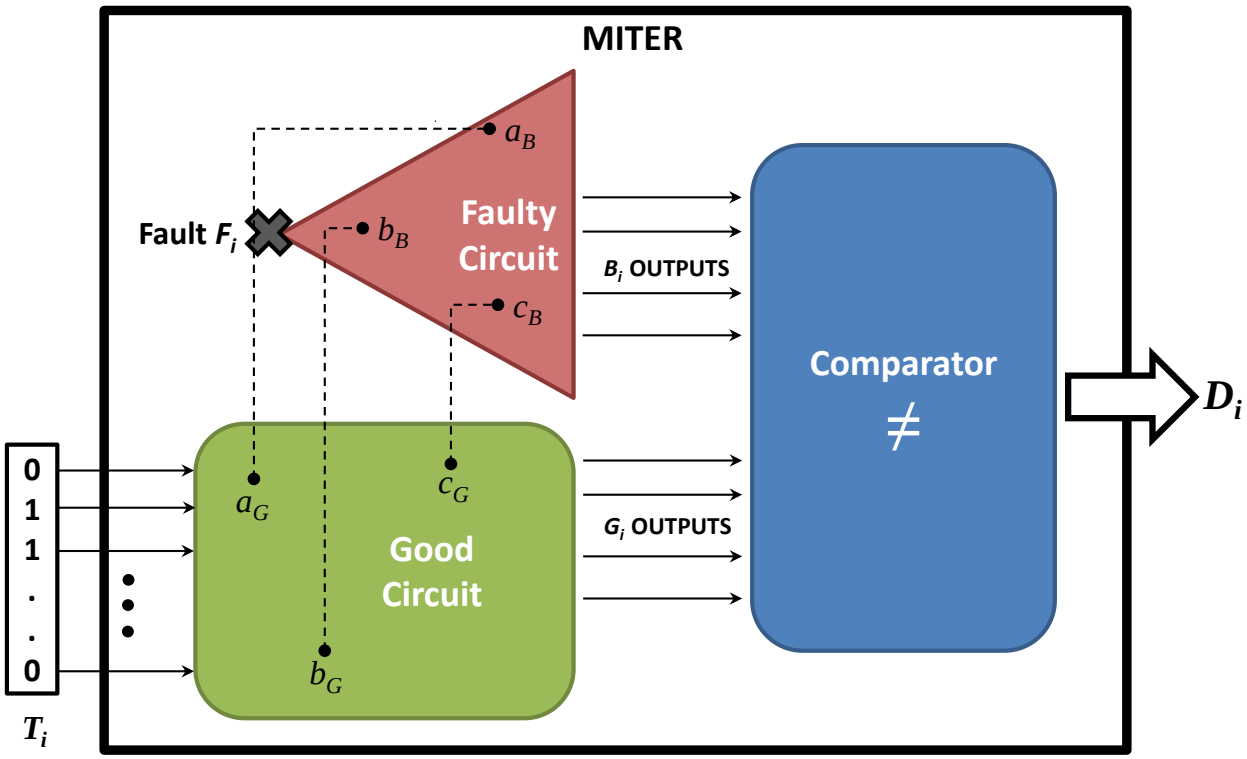


Figure 3.2: Fault detection formula for fault F_i

Further optimizations are made to reduce the number of variables in $\mathbf{B_i}$. We make arrangements to remove variables for gates which do not participate in the *injection* or *propagation* of fault F_i, by pre-computing the fan-in and fan-out *cones of influence* of F_i. Figure 3.2 illustrates the optimized fault-detection formula for fault F_i. The good circuit includes all the gates of the circuit. The faulty circuit includes only the gates

in the fan-out cone of the fault. In addition to the fan-out cone of the fault, *side-input* gates are also included for the faulty circuit. A side-input gate is a gate such that it does not lie in the fan-out cone of the fault, and at least one of its outputs is connected to a gate in the cone. In Figure 3.2 signals a, b and c are outputs of side-input gates for fault F_i. The fault cannot propagate to these signals and thus they should have the same values in the good and faulty circuits. We include constraints to enforce this equality. The miter circuit is driven by the input vector T_i and produces a Boolean output D_i, which is true if and only if T_i detects the fault F_i. As long as all the side-input signals in the faulty-circuit have been constrained to be equal to their good-circuit counterparts, the faulty circuit is implicitly driven by the same input vector T_i. Note that these ideas were proposed in [74], and the formulation for test generation is quite similar to the SAT-based test generation model described therein.

(c) **Constraints to impose $\mathbf{D_i}$ to be *true* for $0 \leq i \leq (K-1)$**

(d) **Constraints to connect every $\mathbf{D_i}$ to the bit-vector variables in C**

These constraints effectively model the mapping between the T_i test-vector variables and the S_i LFSR state vectors. We add constraints to ensure that every test vector T_i is covered by at least one vector that can be generated by the LFSR. Formally, the constraint

$$\bigwedge_{i=0}^{K-1} \left[\bigvee_{j=0}^{L-1} (T_i = S_j) \right]$$

ensures that every bit-vector T_i is equal to one of the S_j variables. Along with the detection constraints $\mathbf{D_i}$, this ensures that every fault in set F is detected within L cycles of the LFSR starting from seed S_0. To further constrain the search space for the SMT solver, we add the constraint

$$\bigvee_{i=0}^{K-1} (T_i = S_1)$$

which ensures that the computed seed detects at least one fault from set F. This constraint helps avoid (and prune from the search space) those variable assignments in which LFSR cycles are 'wasted' due to fault-detection starting at a vector S_i such that $i > 0$.

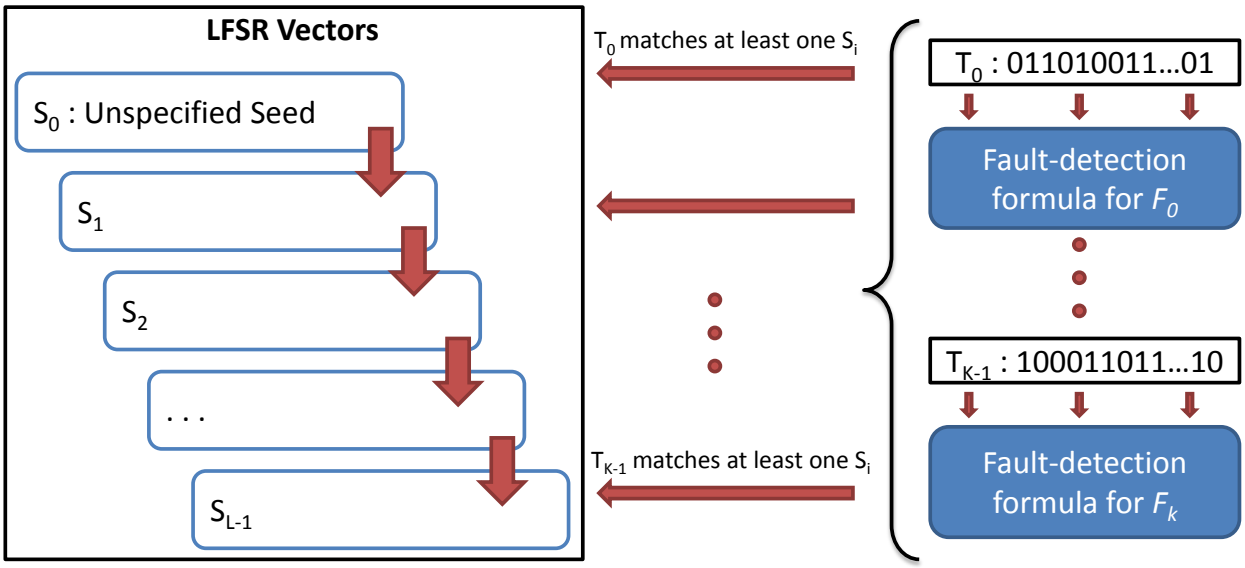


Figure 3.3: Complete SMT formulation

The complete formulation is as illustrated in Figure 3.3. The SMT model is satisfiable if and only if all faults in set F are testable and the *F-Distance* of F is $\leq L$. Hence, the value of S_0 in the satisfying assignment gives a seed that can *chain* the faults in F within L cycles from the computed seed. Also, an *unsat* returned by the solver would indicate that the *F-Distance* of F is greater than L.

3.3 Discussion

The above SMT formulation for *fault-chaining* is the core of our method for seed computation and reduction. It promises significant advantages over previous approaches. First, our method

does not require test-vectors to be pre-determined and pre-ordered, as is frequently required by methods based on solving linear equations. In fact, it does not even impose any restrictions on the order in which faults in set F are to be detected. Since test-vectors bits are not required to be fixed in advance, the method implicitly allows for a single LFSR vector to potentially detect multiple faults. It also allows for 'don't care' cycles between test-vectors in the LFSR cycle, where the LFSR vector does not detect any of the faults under consideration. Consequently, the method is very general and gives us a better chance of finding a seed to cover a given set of faults. Additionally, if the SMT solver returns an *unsat*, it suggests that the faults in F are indeed hard-to-chain. Our method eliminates any need for reordering test-vectors and/or considering alternate vectors for detecting the same faults.

We compare our method with previous work on seed computation using SMT. The work in [63] also attempts to combine the processes of seed computation and test-generation. However, the problem is cast as one big SMT formula resulting in a resource expensive call to the SMT solver. In contrast, the method we propose is more cost-effective as it no longer needs to have L copies of the circuit. In any single call to the SMT solver, we only need as many copies of the circuit as the number of faults to be chained. Greater numbers of faults we want to chain would demand larger SMT formula sizes, and in turn, longer SMT solving times to return a solution.

So far, the new SMT formulation allows us to generate seeds that can chain multiple faults. However, we are faced with a question of which faults to chain. The next chapter presents our fault clustering method to select the faults.

Chapter 4

Seed Computation and Reduction Flow

In this chapter, we present a complete flow for computing a small set of seeds, given a circuit and a list of detectable (irredundant) faults.

4.1 Fault Clustering using Independence Graph

Given the SMT formulation described in Chapter 3, we need to identify the K faults that we wish to chain. Although any set of K faults can be chained, we would like to be more clever about it to reduce the number of seeds obtained at the end. This section describes how we select these faults to be chained. In order to reduce the number of seeds, a seed that can chain those faults currently undetected by any single seed would be helpful. This step is critical to the efficacy and efficiency of our overall method.

The proposed *fault-chaining* method considers K faults at a time within an LFSR 'window' of length L cycles. However, a seed returned by the SMT solver is usually able to detect additional faults. We can determine all faults that a seed covers by first simulating the

LFSR starting from the seed, followed by fault-simulation on the entire fault-list using the LFSR vectors. We perform this fault-simulation without fault-dropping. This helps us determine a list of *all* faults that a seed can cover, even including faults which were not explicitly considered while computing the seed. In essence, we aim to generate additional seed candidates to chain those faults currently not detected by any existing seed. The newly computed seed may render some previously computed seeds unnecessary, thereby reducing the cardinality of the set of seeds.

At every iteration in our method, we start with a certain number of seed candidates in our pool. We simulate the candidates as described above to obtain the faults detected by each candidate. This information is represented in a fault-dictionary, which is a binary matrix. A sample dictionary with 3 seed candidates for a circuit with 5 faults is shown below:

	F_1	F_2	F_3	F_4	F_5
Seed 1	1	0	1	1	1
Seed 2	0	1	1	1	0
Seed 3	1	0	1	0	0

In this example, there are 3 seed candidates for 5 faults. Seed 3 covers faults F_1 and F_3; fault F_4 is covered by Seed 1 and Seed 2. We use this dictionary to determine the sets of faults to be chained in the next iteration of SMT solving. For this purpose, we build a *fault-independence graph*. We define two faults in a dictionary to be *independent* if there currently exists no seed candidate which can detect both faults. We first identify all pairs of independent faults. Note that each column in the dictionary is a bit-vector that represents one fault. We compute the *dot product* of two columns by counting the number of 1s in the bitwise AND of the corresponding bit-vectors. This number gives the number of seed candidates which detect both faults. By computing the dot product for every pair of faults, we identify independent faults (i.e., the fault pairs with dot product = 0).

Using the dot product computed for all fault-pairs in the dictionary, we can now build a fault-independence graph such as the one shown in Figure 4.1. Each node in the graph

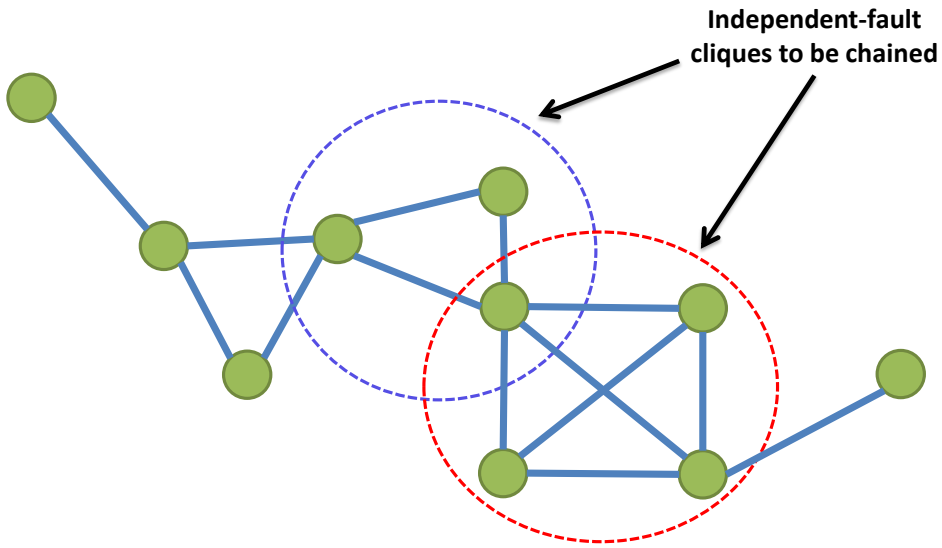


Figure 4.1: Example of fault independence-graph

represents one fault. We add an edge between two faults if and only if their corresponding dot product as computed above is exactly 0. After the independence-graph is built, we identify *cliques* in the graph. Note that a clique of size N in the independence graph represents a set of N faults such that no pair of faults are detected by any seed candidate in the pool. It has been shown in [75] that:

> The size of the largest clique in the independence graph (aka. clique number) is a lower bound on the single-detection test set size.

Generalizing this result, we can see that the clique number of the independence graph shown in Figure 4.1 gives the minimum number of seeds if we were restricted to using only those seed candidates from the current pool. This observation suggests that we are likely to benefit from chaining together those faults which form cliques in the independence graph. Additionally, due to reasons described earlier, we want to limit the sizes of these cliques. Thus, the next step in our method is the partitioning of the independence graph into non-singular cliques of bounded size. The clique partitions need not be mutually independent. Algorithm 4.1 describes the greedy algorithm used to identify cliques from the graph such that every fault

node with a non-zero degree is included in at least one clique.

Algorithm 4.1 Clique-partitioning

Inputs: Independence graph G, Clique size bound B
Output: Set of cliques C

1: $C \leftarrow \phi$
2: **for all** nodes $N \in G$ **do**
3: $Q \leftarrow \phi$
4: **if** N is marked covered OR degree(N) ≤ 1 **then**
5: skip N and go to next node
6: **else**
7: $Q \leftarrow \{N\}$
8: mark N covered
9: $common \leftarrow \{n \mid n$ is a neighbor of $N\}$
10: **while** $|Q| < B$ AND $|common| > 0$ **do**
11: $M \leftarrow$ max degree node from $common$
12: $Q \leftarrow Q \cup \{M\}$
13: $common \leftarrow common \cap \{$neighbors of $M\}$
14: **end while**
15: $C \leftarrow C \cup Q$
16: **end if**
17: **end for**

Note: In line 11, if multiple nodes have max degree, then we pick the one which has been 'covered' least number of times.

4.2 Seed Reduction by Set-covering using ILP

While Chapter 3 describes how seed candidates for fault groups are computed, Section 4.1 describes how groups of faults to be chained are identified. As seed candidates are computed and added to the pool, the fault-dictionary size keeps growing. In every iteration, following fault-simulation of the new seed candidates, we use the dictionary to determine a small subset of seeds that are sufficient to cover all the faults in the circuit. The problem of selection of minimum number of seeds from the dictionary that can detect all faults can be cast as a standard *Set Covering* problem—with the columns being elements to cover and each row representing a subset of elements it covers.

We use an Integer Linear Programming (ILP) formulation for computing the set cover. Modern ILP solvers are quite efficient at solving optimization problems of this type. The formulation has been widely used and discussed in literature. Optimization procedures and LP-solving have been shown to be very effective for test compaction procedures [41–44]. The integer linear program for selecting the set cover of seeds is described below:

$$\text{minimize} \sum_{i=1}^{N_S} x_i$$

under the constraints:

$$\forall \text{ faults } F_j, \left(\sum_{i \in D_j} x_i \right) \geq 1$$

$$\forall i \in \{1 \ldots N_S\}, 0 \leq x_i \leq 1$$

where:

$$\begin{aligned} N_S &= \text{Number of seeds in dictionary} \\ D_j &= \{i \mid \text{Seed } i \text{ covers fault } F_j\} \\ x_i &= \begin{cases} 0 \Leftrightarrow (\text{Seed } i \text{ excluded from solution}) \\ 1 \Leftrightarrow (\text{Seed } i \text{ included in solution}) \end{cases} \end{aligned}$$

4.3 Overall Procedure

We now describe our complete procedure for seed-computation and reduction. We start by fixing values for the following parameters used by the method:

(a) LFSR - We choose an LFSR of length = N_I, the number of primary inputs of the circuit.

(b) LFSR polynomial - We choose a *primitive polynomial* from [76] to yield a maximal-length LFSR.

(c) Clique-size bound B.

(d) LFSR window size of L cycles for SMT based chaining.

(e) Fault-simulation window length of M cycles - This is the number of cycles the LFSR will be run for, after loading each new seed.

The overall flow for computing a small set of sufficient seeds is illustrated in Figure 4.2. We start by identifying the detectable faults and remove the undetectable ones. For each detectable fault, a test-vector is also generated. Once a complete set of test-vectors is obtained, we assume this set to be our initial set of seed candidates and build a seed-fault dictionary as described in Section 4.1, by simulating the LFSR for M cycles for each seed candidate.

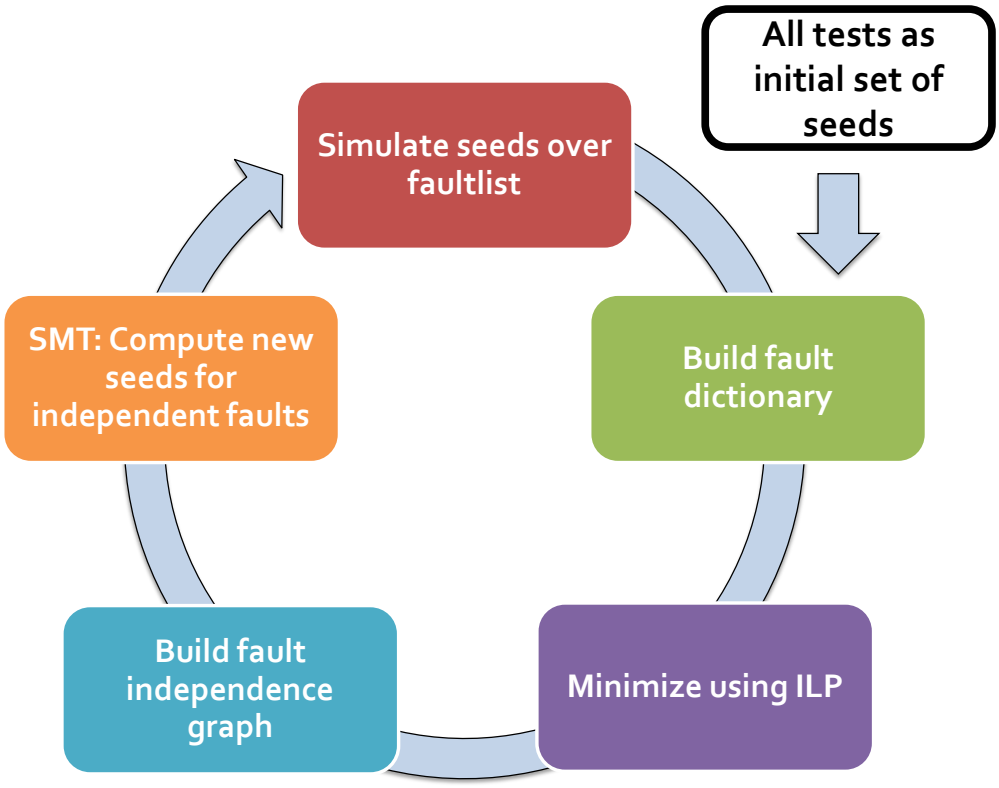


Figure 4.2: Iterative seed-reduction procedure

Next, we use an ILP solver to select minimum number of seeds from the dictionary to cover all faults. This is done using the set-cover formulation described in Section 4.2. The size of

the set-cover solution gives the first estimate on the solution quality. This initial set would likely be far from optimal. However, we use this set to compute independent faults.

In order to chain currently unchained faults into a common seed, we build the independence graph and partition it into cliques of size $\leq B$ as described in Section 4.1. SMT solving as described in Chapter 3 gives us one seed for every clique of faults chainable within L cycles of the LFSR. We add the new seeds to our current pool of seed candidates, and simulate these seeds to augment our fault-dictionary. Note that in any iteration, we do not replace the seeds from the previous iterations. Since we keep adding new (possibly better) seeds to the pool, the size of the solution across iterations monotonically decreases. We can repeat this iterative procedure until either a sufficiently low number of seeds is selected by ILP, no edges exist in the independence graph, or the number of seeds remains constant for a number of iterations.

4.4 Experiments and Results

We have experimentally evaluated the proposed method by running it for the ISCAS'85, ISCAS'89 and ITC'99 benchmark circuits, as well as some circuits from OpenCores. Full-scan versions of sequential circuits were used. The SMT solver used for the fault-chaining step was Z3 v3.2 [32]. The optimizer used for solving the set-covering ILP was Gurobi v5.0.1 [38]. All experiments were performed on an Ubuntu Linux workstation with an Intel ® Core ™ i7 3.33 GHz CPU, and 6 GB of memory.

We compare the results of our method against the optimum results from random seeds, as well as the optimum results from our initial pool of seeds. First, we generate 1000 random seeds, simulate the LFSR for a number of cycles followed by fault-simulation as described earlier to build a fault-dictionary. We minimize this dictionary using ILP to determine the minimum number of random seeds sufficient to cover all faults. We only perform our experiments on circuits for which random seeds do not give a satisfactory solution. Table 4.1 reports the

Table 4.1: "Easy" benchmark circuits

Circuit	# PIs (N_I)	# Faults	M	# Sufficient random seeds
c432	36	520	1k	1
c499	41	750	500	1
c880	60	942	3k	1
s953f	45	1079	2k	3
s1423f	91	1501	5k	2
c1355	41	1566	3k	1
c1908	33	1870	2k	2
c3540	50	3291	3k	2
s5378f	214	4563	5k	4
c5315	178	5291	3k	1
c6288	32	7710	500	1
b03	34	374	100	1
b04	77	1472	3k	2
b05	35	1928	3k	2
b08	30	433	3k	1
b09	29	415	1k	2
b10	28	478	1k	1
b11	38	1267	3k	2

"easy" circuits for which it is very easy to find a small number of random seeds for complete fault coverage. These circuits are largely random-pattern testable, making LBIST specially applicable to them.

Table 4.3 documents results and performance of the overall seed-computation procedure described in Section 4.3. Column 3 denotes the total number of detectable single-stuck-faults in the circuit. The parameter M in column 4 is the size of the fault-simulation window, and is chosen empirically based on the value of N_I. We report three sets of results relevant to our method. Column 5 reports the minimum number of seeds required from 1000 random seeds to cover all faults using ILP solving. Blank entries in this column indicate that 1000 random seeds were not able to achieve complete fault coverage. Next, we use our ATPG tool to generate tests for the entire fault-list. Each of the tests is then assumed to be an LFSR seed. This set of seeds is simulated over M cycles and reduced like the set of random seeds just described. Columns 6 and 7 report the number of tests and the required number of seeds, respectively. Note that this is the starting point for the iterative method described in Section 4.3. The last 3 columns report, for our overall procedure, the number of iterations, total runtime and size of the final solution (number of seeds). When running this experiment, the method ends when the number of seeds does not reduce for 3 consecutive iterations. In these experiments we have set the clique-size bound B to 3 and the LFSR window size for SMT (L) to 100. If the SMT-solver returns *unsat* for more than 90% of the number of cliques to be chained, L is increased by 50 and the SMT solver is called again. Similar experiments were conducted for the ITC99 benchmark circuits as well. The results are reported in Table 4.4. The table format is similar to Table 4.3.

Note that some of the circuits have significantly large number of faults. Some characteristics of these "large" circuits are summarized in Table 4.2. We report the number of primary inputs, gates and faults in the circuits. Before running our seed reduction procedure, we select a subset of faults that are 'hard-to-test'. For this purpose, we simulate the LFSR for M cycles from each of 1000 random seeds and remove all the faults which were detected by each one of the 1000 seeds, marking them as 'easy' faults. The number of faults that

Table 4.2: Characteristics of large benchmark circuits

Circuit	# PIs	# Gates	# Faults	# HTT-F
s13207f	700	9443	9664	2196
s15850f	611	11069	11336	1273
b14f	275	4967	12599	1970
b15f	485	9275	22740	7982
b20f	522	9912	24852	4469
b21f	522	10296	26084	3967
b22f	767	15809	39488	6253
spi	277	5600	10900	1375
tv80	372	12035	25807	7109
systemcaes	929	17872	34637	5614

remained are reported in the HTT-F column in Table 4.4. This faultlist reduction helps reduce the computational cost of our method significantly, by allowing us to work with a smaller fault-independence graph and a smaller fault-dictionary.

It can be observed from Tables 4.3 and 4.4 that the method produces a solution with very few number of seeds, with a very low runtime. Most of the final solution sizes are a single-digit number of seeds. The total number of bits to be stored on-chip may be calculated by multiplying the solution size by N_I. The difference between the number of seeds before and after the method (columns 7 vs. 10) demonstrates the strength of fault-chaining. For example, for c7552, our method is able to achieve a seed-reduction of over 78% with our fault-chaining. The results are similar when compared to random seeds as well.

In Table 4.5, we compare our results to recent work on LFSR reseeding using SMT solving. In [63], a random seed was first simulated for 1000 cycles and all the detected faults dropped. SMT-based seed computation was then performed for the remaining faults, using an LFSR window size of $n + 20$, where n is the number of faults under consideration reported in Column 3. To make the comparison fair, we use the same value for LFSR-window ($L = n+20$). Additionally, we set the fault-simulation window size M equal to L. Even with virtually

Table 4.3: Experimental results on ISCAS benchmark circuits

Circuit	# PIs (N_I)	# Faults	M	1k Rand Seeds	All tests as seeds		Fault chaining		
				ILP Sol.	#Tests	ILP Sol.	# Iterations	Runtime(s)	Sol.
s420f	35	430	1k	-	70	8	4	12	**3**
s641f	54	467	2k	8	85	7	4	27	**3**
s713f	54	543	2k	6	85	9	4	23	**3**
s832f	23	856	1k	**4**	175	5	3	14	**4**
s838f	67	857	3k	-	129	34	7	434	**7**
s1196f	32	1242	1k	7	198	7	5	157	**5**
s1238f	32	1286	1k	**6**	222	8	3	164	**6**
c2670	233	2630	10k	18	150	29	3	646	**4**
s9234f	247	6475	10k	-	592	31	11	4734	**17**
c7552	207	7419	10k	-	269	37	5	4207	**10**
s13207f	700	2196	10k	**6**	449	**6**	3	1568	**6**
s15850f	611	1273	10k	-	388	66	12	25417	**19**

Table 4.4: Experimental results on ITC99 and OpenCores circuits

Circuit	# PIs (N_I)	# Faults	M	1k Rand Seeds	All tests as seeds		Fault chaining		
				ILP Sol.	#Tests	ILP Sol.	# Iterations	Runtime(s)	Sol.
b07f	50	1100	1k	5	73	7	5	61	**3**
b12f	126	2766	3k	13	199	13	4	825	**6**
b14f	275	1970	10k	-	497	219	18	43534	**74**
b15f	485	7982	10k	-	523	82	9	48230	**33**
b20f	522	4469	10k	-	533	303	19	99123	**133**
b21f	522	3967	10k	-	453	333	16	77157	**132**
b22f	767	6253	10k	-	703	286	9	71922	**100**
spi	277	1375	10k	25	685	24	6	7296	**19**
tv80	372	7109	10k	-	1111	93	8	34954	**43**
systemcaes	929	5614	5k	12	251	13	9	7670	**9**

everything else being the same, the proposed method is able to produce better or equal results in all cases, with a performance improvement of multiple orders of magnitude. As discussed earlier, this may be attributed to the fact that [63] relies on a few very expensive calls to the SMT solver. The method we have proposed aims to make the problem tractable by trying to chain together those faults which are likely to give the greatest benefit.

Table 4.5: Experiments: Comparison with [63] (Data used under fair use, 2012)

Circuit	# Faults	# Inj. (n)	M	[63]		Fault chaining	
				Time	Sol.	Time	Sol.
c880	942	21	41	6071	**3**	4	**3**
c1355	1566	28	48	30917	**4**	22	**4**
c1908	1870	84	104	73002	12	117	**7**
s953f	1079	117	137	183809	15	891	**7**
s1423f	1501	38	58	92346	9	78	**6**
s1488f	1486	40	60	159219	13	45	**8**

Our method efficiently integrates simulation with analysis. Simulation helps us reason about faults detected by using a seed for more number of cycles. The number of seeds to be stored may be further reduced by using a larger value for parameter M. This observation allows our method to offer a trade-off between seed-storage overhead and test-length to cover all faults.

Chapter 5

Test-Point Insertion for Further Seed Reduction

In Chapters 3 and 4 we described a procedure to compute a set of LFSR seeds to cover all detectable faults in the circuit. In this chapter, we will discuss methods to improve the testability of the circuit using test-points, with the goal of reducing the seeds further. Our method first identifies signals for control-point insertion, followed by observation points.

5.1 Selection of Faults for Testability Enhancement

The methods we will propose for identifying signals for test-point insertion rely on detailed fault-simulation. For this reason, we would like to first reduce the size of fault-list under consideration. The idea is to select a subset of faults such that inserting test-points to aid in their detection is likely to give the greatest benefit for seed reduction.

We now describe an ILP formulation for a constrained version of the set-cover formulation described in Section 4.2. Running an ILP solver on this formulation will help us identify a subset of faults for test-point insertion. Given a seed-fault dictionary like the one described

in Section 4.1 and an integer S_{max}, we formulate the following optimization model to identify a set of no more than S_{max} seeds from the dictionary which achieves the maximum combined fault-coverage.

$$\text{maximize} \sum_{j=1}^{N_F} C_j$$

under the constraints:

$$\forall \text{ faults } F_j, \left(\sum_{i \in D_j} x_i \right) - C_j \geq 0$$

$$\left(\sum_{i=1}^{N_S} x_i \right) \leq S_{max}$$

$$\forall i \in \{1 \ldots N_S\}, 0 \leq x_i \leq 1$$

$$\forall j \in \{1 \ldots N_F\}, 0 \leq C_j \leq 1$$

where:

$$\begin{aligned}
N_F &= \text{Number of faults in dictionary} \\
N_S &= \text{Number of seeds in dictionary} \\
D_j &= \{i \mid \text{Seed } i \text{ covers fault } F_j\} \\
x_i &= \begin{cases} 0 \Leftrightarrow (\text{Seed } i \text{ excluded from solution}) \\ 1 \Leftrightarrow (\text{Seed } i \text{ included in solution}) \end{cases} \\
C_j &= \begin{cases} 0 \Leftrightarrow (\text{Fault } F_j \text{ is covered in solution}) \\ 1 \Leftrightarrow (\text{Fault } F_j \text{ is not covered in solution}) \end{cases}
\end{aligned}$$

In this formulation, x_i are pseudo-Boolean variables that indicate the inclusion or exclusion of seed i from the solution. C_j are indicator variables representing whether fault F_j is covered by any of the seeds included in the solution. The first constraint ensures that a C_j variable will be assigned a *true* value **only if** at least one of the seeds that cover fault F_j is also

included in the solution. This ensures that a fault will never be wrongly marked as *covered*. In cases where the ILP solver does not prove an optimal solution due to a timeout, a fault may still be wrongly marked as *not-covered*. However, we are prepared to accept this error, since it will be automatically rectified when fault simulation is performed in the next step. The variable assignments in the solution not only give us the exact set of faults covered (or equivalently, missed) by S_{max} seeds, they also give us the exact subset of seeds selected. These seeds are noted to be the best S_{max} seeds in the dictionary.

Using the formulation described above and the dictionary from the last iteration of the SMT-based seed computation method of Chapter 4, we call the ILP solver with multiple values of S_{max} (typically all values from 1 up to around 10). We generate a database of number of seeds S vs. the number of faults missed. A suitable number S_{SEL} of the best seeds is selected empirically from this database. S_{SEL} is typically chosen to be as small as possible while being large enough that the number of faults missed is not too great (which would cause fault-simulation time to increase).

We generate the LFSR patterns that these S_{SEL} seeds will produce to get a test-set comprising MxS_{SEL} patterns (where M is the number of cycles each seed is to be run for). Detailed fault-simulation is run using these test patterns for the faults in F_{UND} (the set of faults missed). Each fault in F_{UND} belongs to one of three cases:

(I) The fault is not excited by any of the vectors enumerated from the S_{SEL} seeds.

(II) The fault is at the input of gate G (aka. *branch fault*), and is excited but not propagated to the output of G. In other words, the fault is excited but is blocked due to controlling-values on the side-inputs of G.

(III) The fault is excited and propagated to the output of one or more gates in the circuit, but was never detected.

We call the faults belonging to cases I and II collectively as set F_{CTRL}, the set of faults that are undetected/unexcited due to controllability issues. The faults in III are undetected due

to observability and possibly controllability issues. We focus on F_{CTRL} for control point insertion.

5.2 Control-Point Insertion

We will now describe our method to identify candidate locations for control-points to target faults in the set F_{CTRL}, followed by our proposed control-point implementation.

5.2.1 Identification of Control-Point Sites

The first step in our control-point identification procedure is to perform testability analysis. We compute COP values ($C0$ and $C1$) for all gates according to the rules described in Section 2.1.3. We will use the COP values to guide our backtracing algorithm for identifying candidate control-points. For any signal, $C0 + C1 = 1$. The closer the COP value of a signal is to 0.5, the easier it is to control. We decide a threshold COP value above which a signal is termed as **hard-to-control**. A signal for which MAX($C0$,$C1$) is greater than the threshold is a hard-to-control signal.

We propose a backtracing + scoring procedure for identifying control-point candidate sites. The backtracing is partly based on the source tracking algorithm presented in [73]. The key idea behind backtracing is the observation that there are two kinds of gates for which the output signal is hard-to-control:

(a) **Gates with one or more hard-to-control inputs:** For example, the AND gate shown in Figure 5.1 might have a very low value of $C1$ due to two of its inputs being hard-to-control. Similarly, the OR gate might have a low $C0$ value.

(b) **Gates with large fan-in:** For example, the gates shown in Figure 5.2 might have hard-to-control outputs even though all their inputs might be easy-to-control.

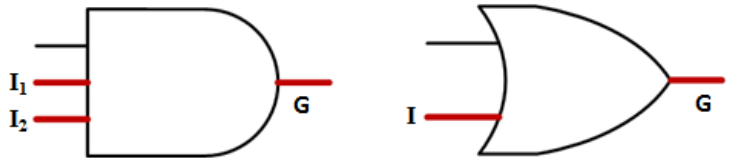


Figure 5.1: Gates with hard-to-control inputs (Bold red signals are hard-to-control)

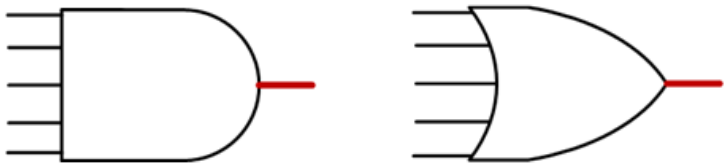


Figure 5.2: Gates with large fan-in (Bold red signals are hard-to-control)

Algorithm 5.1 describes the backtracing and scoring algorithm. Given a fault location, we first initialize Q as the list of signals to initiate backtracing from. We maintain a global score for each signal. We assign higher scores to signals which are likely to be stronger control-point candidates. We start by backtracing from the fault site, which includes:

- Signal F itself, if the fault is at the output of gate F.
- The side input signals A and B, if the fault is at the input of gate F.

These cases are illustrated in Figure 5.3.

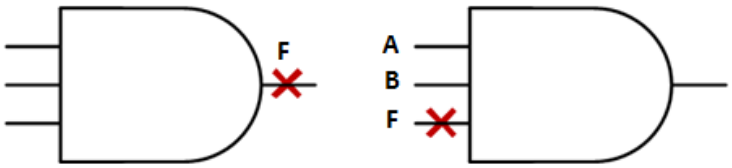


Figure 5.3: Backtrace Initialization

The backtrace procedure progresses by looking at the inputs of a hard-to-control gate G and counting the number of inputs that are hard-to-control ($hCount$). If the gate G has only one input that is hard-to-control, we do not increment the score for G and proceed to backtrace

through the input I of G. Gate G is effectively skipped. An example would be the OR gate of Figure 5.1. We would like to control I rather than G, since it would have the same effect on the controllability of G, and by controlling I we might be able to affect more nodes in the circuit compared to controlling G alone.

For gates such as the AND gate in Figure 5.1 which have multiple hard-to-control inputs, we use the formula stated on line 20 of the algorithm to increment the score for G. The reasoning behind the formula is that a gate becomes a stronger candidate if it has a higher fan-in. It becomes an even stronger candidate if it has multiple hard-to-control inputs. Note that we maintain a global score for each gate and increment it during the backtrace procedure. The reason for this is to account for the possibility of a gate affecting the fault-site through multiple paths. Intuitively, this would make the gate a stronger control-point candidate. In our algorithm, due to the recursive implementation of backtrace, such a gate would be traced through multiple times, accomplishing out objective of doubling its score.

After performing backtrace and scoring for each fault in F_{CTRL}, we add up the scores for each gate across all the backtraces. Now we may select the desired number of control points with the highest scores for control-point insertion.

5.2.2 Control-Point Implementation

Once we have selected the signals for control-point insertion, we are ready to insert control-points into the circuit. We propose the insertion of a control-point in the form of an *inversion-point*, using a single XOR gate for each point as shown in Figure 5.4. In addition to the XOR gates, one extra signal CM (Control Mode) is introduced which is common for all control-points. When CM = 0, the XOR gates become transparent and the circuit resumes normal operation. When CM = 1, the XOR gates act as inverters and the values of all the controlled-points inverts simultaneously. In the figure, signals P and Q are control-point candidates, and their controlled versions are P_{ctrl} and Q_{ctrl}. The advantages of this kind of inversion point over conventional Control-1 or Control-0 points are:

Algorithm 5.1 Backtracing + Scoring for Control-Point Identification

Initialize: $Q = \{$Candidates at fault-location$\}$
do BACKTRACE(G, 0) forall gates $G \in Q$

Procedure: BACKTRACE
Inputs (gate G, level L)
Output Updates gate scores in SCORE[ ]

```
1: if L = LevLimit then
2:    return
3: end if
4: if fanin[G] = 1 then
5:    I = input of G
6:    BACKTRACE(I,L)
7: else if fanin[G] = 2 then
8:    btList = { hard-to-control inputs of G }
9:    hCount = | btList |
10:   if hCount = 2 or hCount = 0 then
11:      SCORE[G] += 2
12:      L += 1
13:   end if
14:   for all gates I ∈ btList do
15:      BACKTRACE(I,L)
16:   end for
17: else
18:   btList = { hard-to-control inputs of G }
19:   hCount = | btList |
20:   SCORE[G] += (hCount + fanin[G] −2)
21:   L += 1
22:   for all gates I ∈ btList do
23:      BACKTRACE(I,L)
24:   end for
25: end if
```

- Ease of implementation
- Low hardware overhead of one XOR gate per point in addition to a single additional control signal
- The control point acts as Control-1 point for signals hard-to-control to a 1 and Control-0 point for signals hard-to-control to a 0.

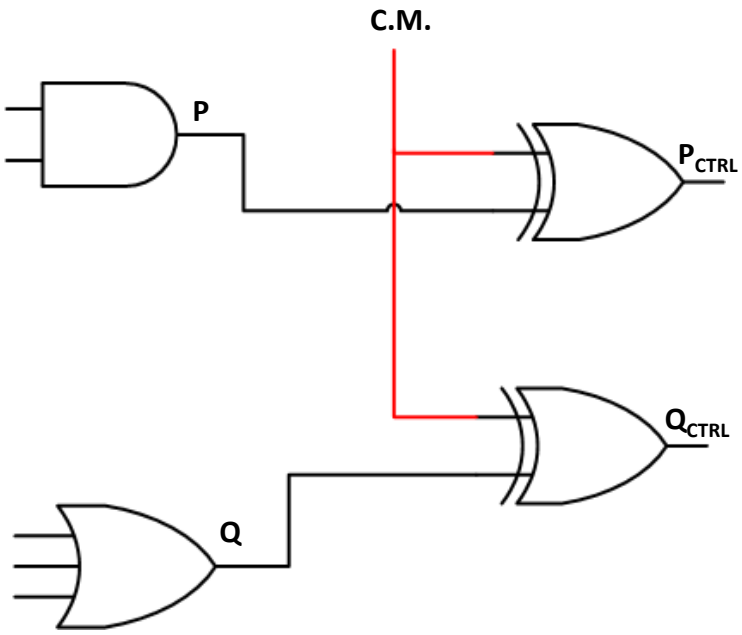


Figure 5.4: Control-point implementation using inversion points

We propose a testing scheme where LFSR patterns from the stored seeds are applied twice to the circuit, once in the normal mode and once in the control mode (with CM = 1). We shall show in the experimental results that a small set of seeds can now detect a significant number of additional faults when run in the control mode, with a small number of signal values flipped by the control-points. Note that other ways to control the CM signal may be possible where two separate test-phases are no longer required. For example, the CM signal may be activated randomly by connecting to one of the LFSR outputs or an unbiased signal within the circuit. Additionally, detailed analysis may be done to identify the patterns in which the

inversion-points are most effective at detecting additional faults, and pattern-decoding logic may be synthesized to control the CM signal in a specific manner.

5.2.3 Observation-Point Insertion

After the control-points have been inserted, we simulate the modified circuit and run fault simulation using the vectors for the S_{SEL} seeds selected earlier. The test patterns are again simulated twice, with CM = 0 and CM = 1. The same patterns now might excite and detect additional faults. We identify the faults that have been excited and propagated to the output of a gate (category III in previous discussion). For each such fault, we make a list of all the points in the circuit that the fault was found to have propagated to. These points are observation point candidates for the fault. If any of these signals were made observable, the S_{SEL} seeds would now detect the fault.

We build a dictionary similar to the seed-fault dictionary, with the seeds now being replaced by observation-point candidates. Next we use the maximum set-cover formulation described in Section 5.1 to select a subset of the observation points which together help in detecting the most number of faults. We then insert observation points on these candidate signals. We can implement observation points in one of many ways such as adding a dedicated scan-cell, sharing an existing scan-cell using a MUX etc. Our method is independent of the exact implementation of observation points, and it will work as long as the point is made observable at the output in some way.

After both control and observation points have been inserted, we run the SMT-based chaining method described in Chapter 4 on the modified circuit. In each iteration, during fault-simulation of new seeds, LFSR patterns from each seed are simulated for both CM = 0 and CM = 1. The Control Mode signal is an unconstrained variable in the SMT formulation. This means that every fault-detection formula is free to use either value of CM. This is helpful, since it ensures fault chainability regardless of whether the tests were run in control-mode or normal mode. The entire flow for testability enhancement is illustrated in Figure 5.5.

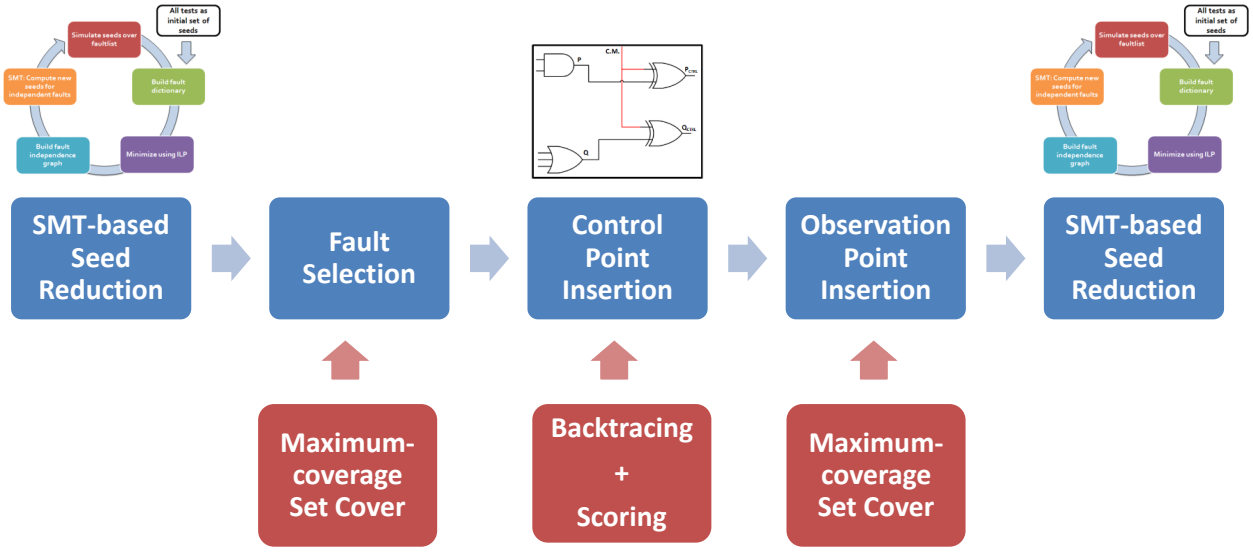


Figure 5.5: Seed reduction using test-point insertion

5.3 Experiments and Results

Experiments were performed on the benchmark circuits to reduce the seeds further after the method described in Chapter 4. We take those circuits for which the number of seeds was significantly high even after SMT-based chaining and reduction. Starting from the results of Chapter 4, we go through the test-point insertion process illustrated in Figure 5.5. We maximize the coverage for a small number (S_{SEL}) of seeds, and simulate these seeds to identify the set of faults ($|F_{CTRL}|$) to be considered for control-point insertion. Next we identify and score potential control-point locations as described in Section 5.2, using the backtracing and scoring algorithm. After inserting control-points, we identify the best observation points as described in Section 5.2.3. After we have the modified circuit with the control and observation points, we run the SMT-based seed reduction flow again. Table 5.1(a) reports the results for ISCAS circuits after test-point insertion.

In Table 5.1(a), Column 2 reports the number of seeds that were selected by the SMT-based method in Chapter 4. Column 3 reports the number of seeds that were selected from S_{SMT} for fault-simulation, as described in Section 5.1. $|F_{UND}|$ in Column 5 reports the number of

Table 5.1: Test-point insertion results on benchmark circuits

(a) Test-point insertion results for ISCAS circuits

Circuit	$\|S_{SMT}\|$	S_{SEL}	# Faults	$\|F_{UND}\|$	$\|F_{CTRL}\|$	# CP	# OP	$\|S_{TPI}\|$	Reduction
s832f	4	1	856	60	1	1	2	3	25%
s838f	7	1	857	61	42	6	5	4	42.86%
s1196f	5	1	1242	68	6	7	2	3	45%
s1238f	6	1	1286	91	7	10	5	3	50%
s9234f	17	1	6475	519	84	10	5	4	76.47%
c7552	10	1	7419	94	12	13	5	2	80%
s13207f	6	1	2196	468	254	10	5	5	16.67%
s15850f	19	1	1273	454	98	6	7	9	52.63%

(b) Test-point insertion results for ITC99 circuits

Circuit	$\|S_{SMT}\|$	S_{SEL}	# Faults	$\|F_{UND}\|$	$\|F_{CTRL}\|$	# CP	# OP	$\|S_{TPI}\|$	Reduction
b12f	6	1	2766	63	9	3	3	2	66.67%
b14f	74	4	1970	764	23	5	5	15	79.73%
b15f	33	5	7982	1032	39	5	5	8	75.76%
b20f	133	5	4467	1489	6	5	5	41	69.17%
b21f	132	3	3967	1544	36	5	5	35	73.48%
b22f	100	5	6253	1416	14	5	5	37	63%

faults that were missed by the best S_{SEL} seeds, while $|F_{CTRL}|$ is the number of faults out of $|F_{UND}|$ that were considered for control point insertion. Next, we report the number of inserted control-points (# CP) and observation-points (# OP), followed by the number of seeds ($|S_{TPI}|$) in the final solution after running the SMT-based reduction method on the testability-enhanced circuit. Table 5.1(b) reports the results after test-point insertion on the larger of the ITC99 circuits. The format of the table is identical to Table 5.1(a).

The last column reports the relative reduction obtained in the number of seeds as a result of the inserted test-points. Note that we are able to achieve a significant reduction with just a few test points. For example, for circuit c7552 we are able to achieve a reduction of 80% in the number of seeds, with the insertion of just 13 control-points and 5 observation points! 2 seeds (instead of 10) are now sufficient to cover all 7419 faults in the circuit. Similar results are noted for some of the largest of ITC99 circuits as well. Similarly for some of the largest ITC99 circuits, we are able to achieve seed reduction in the range of 63-80% by adding no more than 10 total test-points!

These results confirm the efficacy of our test-point selection flow, and reaffirm the popular idea that test-point insertion and testability enhancement can go a long way in improving the efficiency of LBIST. The greatly reduced number of seeds help not only in reducing the on-chip data storage requirements, but also directly affect the test-length and test-time as well.

Chapter 6

Conclusion and Future Work

Design for Testability (DFT) methods such as Logic Built-In Self Test (LBIST) are gaining importance in order to reduce dependence on external testing. In this thesis, we introduced the problem of LFSR reseeding and discussed its significance to LBIST. We have presented methods for computing a small set of seeds for the effective utilization of Linear Feedback Shift Registers (LFSRs) for deterministic testing. Reduction of the number of seeds is important to reduce on-chip storage requirements and consequently the hardware overhead of LBIST.

Most previous methods for LFSR reseeding based test-compression work by encoding pre-computed test patterns. In modern circuits where the number of ATPG patterns may be extremely large, a large number of seeds may still be required. We propose a new way to look at the problem of reseeding, where a seed covers a subset of faults rather than a set of pre-computed patterns. We described a Satisfiability Modulo Theories (SMT) formulation to compute seeds to 'chain' a set of faults. We show that it is convenient to model the combined problem of test-generation and seed computation in SMT, given the expressive power of the theory of bit-vectors.

We presented a complete flow for computing a small set of LFSR seeds to cover all detectable stuck-faults in a circuit. The method works by iteratively identifying groups of faults to

be chained into a common seed, and then using the SMT formulation to compute a new seed candidate. Since we try to chain faults instead of pre-computed test vectors, it helps us overcome most of the problems of vector ordering, grouping and linear-equation solving that traditional seed computation methods have faced. Additionally, we have efficiently integrated simulation and analysis to get the most out of each seed. Tweaking various parameters in the flow can help achieve trade-off between test-length and seed-storage overhead. An optimal number of seeds are selected from the seed candidates using Integer Linear Programming (ILP) at every iteration. The method is general enough that it can be used for any LFSR structure or configuration, coupled with one of many different BIST architectures. Experimental results show that the method is able to produce a very small set of seeds with low computational cost. We showed that the method is able to identify high-quality seeds that can cover many hard-to-test faults and which are virtually impossible to find using random simulation.

Additionally, we proposed techniques to enhance the testability of the circuit via test-points, with the objective of seed reduction. We presented techniques to identify sites to insert control-points and implement them with a very low hardware overhead using logic inversion, to help us make effective reuse of seeds to cover additional faults. Coupled with our method to identify effective observation points, we are able to achieve impressive reduction in the number of seeds. With just a few test-points, we can reduce the already compact set of seeds from the SMT-based chaining method by upto 80%.

Future Work

We now describe possible future directions to extend our work:

- As previously mentioned, our method can be extended to other LBIST architectures. Though we have described SMT-based test generation and seed computation for a test-per-clock scheme, changes may be made in the formulation to use it for test-per-scan schemes as well. By reducing the size of the LFSR for test-per-clock using schemes

such as the one shown in Figure 2.6(b) or test-per-scan schemes such as STUMPS (Figure 2.5), great reductions in the volume of seed-storage may be achieved.

- The SMT formulation may be improved by adding constraints to reduce the search space of the SMT solver. These constraints may include signal-value correlations and necessary signal-value assignments for fault-detection. These optimizations may, in turn, make it possible to use SMT-solving to compute seeds for larger sets of faults at a time (larger cliques) to obtain possibly better results.

- Since SMT can help us conveniently combine the domains of Boolean reasoning and high-level theories, it is possible to adapt our method for test-generation for RTL using word-level constructs instead of bit-level constructs for a gate-level netlist.

- When the size of the fault-list is very large, our method is sometimes ineffective due to the existence of an overwhelming number of cliques. The choice of which cliques to consider becomes an important factor in these cases. Work may be done to reduce the fault-list size and/or come up with better methods to find cliques which can identify 'hard-to-chain' faults in advance.

- Our methods to identify test-point sites may be improved by experimenting with different algorithms for scoring. Our dependence on exact fault-simulation may be removed by the use of previous work done on approximate fault-simulation and methods to estimate the effectiveness of test-point insertion at a given signal.

- In our method, we activate all control points simultaneously in a special testing phase. This might not be the optimum way to utilize control-points. The method may benefit from the use of implementations which allow different test-points to be activated in multiple phases. Also, our Control Mode signal CM may be driven by existing signals by the use of pattern-decoding logic. This may help eliminate the need for additional test mode for each seed and reduce test-times significantly.

Bibliography

[1] International Technology Roadmap for Semiconductors, "ITRS 2008 Update." http://www.itrs.net/Links/2008ITRS/Home2008.htm.

[2] G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. S. M. Hassan, and J. Rajski, "Logic BIST for large industrial designs: Real issues and case studies," in *ITC*, pp. 358–367, IEEE Computer Society, 1999.

[3] M. Bubna, K. Roy, and A. Goel, "HBIST: An approach towards zero external test cost," in *VTS*, pp. 13–18, IEEE, 2012.

[4] V. D. Agrawal, "Editorial - special issue on partial scan design," *J. Electronic Testing*, vol. 7, no. 1-2, pp. 5–6, 1995.

[5] K.-T. Cheng and V. D. Agrawal, "A partial scan method for sequential circuits with feedback," *IEEE Trans. Computers*, vol. 39, no. 4, pp. 544–549, 1990.

[6] C. Fagot, O. Gascuel, P. Girard, and C. Landrault, "On calculating efficient LFSR seeds for built-in self test," in *Proc. European Test Workshop*, pp. 7 –14, may 1999.

[7] E. J. McCluskey, "Verification Testing - A Pseudoexhaustive Test Technique," *IEEE Trans. Computers*, vol. 33, no. 6, pp. 541–546, 1984.

[8] E. J. McCluskey and S. Bozorgui-Nesbat, "Design for autonomous test," *IEEE Trans. Computers*, vol. 30, no. 11, pp. 866–875, 1981.

[9] P. H. Bardell, W. H. McAnney, and J. Savir, *Built-in test for VLSI: Pseudorandom techniques*. New York, NY, USA: Wiley-Interscience, 1987.

[10] L. Wang, C. Wu, and X. Wen, *VLSI Test Principles and Architectures: Design for Testability*. Morgan Kaufmann, 2006.

[11] S. Golomb, *Shift register sequences*. Holden-Day, 1967.

[12] E. B. Eichelberger and E. Lindbloom, "Random-Pattern Coverage Enhancement and Diagnosis for LSSD Logic Self-Test," *IBM Journal of Research and Development*, vol. 27, pp. 265–272, 1983.

[13] H. D. Schnurmann, E. Lindbloom, and R. G. Carpenter, "The weighted random test-pattern generator," *IEEE Trans. Computers*, vol. 24, no. 7, pp. 695–700, 1975.

[14] J. A. Waicukauski, E. Lindbloom, E. B. Eichelberger, and O. P. Forlenza, "A method for generating weighted random test pattern," *IBM J. Res. Dev.*, vol. 33, pp. 149–161, Mar. 1989.

[15] H.-J. Wunderlich, "Multiple distributions for biased random test patterns," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, pp. 584 –593, jun 1990.

[16] D. Kagaris, S. Tragoudas, and A. Majumdar, "On the use of counters for reproducing deterministic test sets," *IEEE Trans. Computers*, vol. 45, pp. 1405 –1419, dec 1996.

[17] C. Fagot, O. Gascuel, P. Girard, and C. Landrault, "A ring architecture strategy for BIST test pattern generation," in *Proc. Asian Test Symp.,*, pp. 418 –423, dec 1998.

[18] C. Dufaza, C. Chevalier, and L. Lew Yan Voon, "LFSROM: A hardware test pattern generator for deterministic ISCAS85 test sets," in *Proc. Asian Test Symp.*, pp. 160 –165, nov 1993.

[19] G. Edirisooriya and J. Robinson, "Design of low cost ROM based test generators," in *Proc. VLSI Test Symp.*, pp. 61 –66, april 1992.

[20] M. Bushnell and V. Agrawal, *Essentials of electronic testing for digital, memory, and mixed-signal VLSI circuits*, vol. 17. Kluwer Academic Publishers, 2000.

[21] N. Jha and S. Gupta, *Testing of digital systems*. Cambridge Univ Pr, 2003.

[22] P. H. Bardell and W. H. McAnney, "Self-Testing of Multichip Logic Modules," in *ITC*, pp. 200–204, IEEE Computer Society, 1982.

[23] B. Könemann, "Built-in logic block observation techniques," in *Proc. 1979 IEEE Test Conf.*, pp. 37–41, 1979.

[24] L. Wang and E. McCluskey, "Concurrent Built-in logic block observer (CBILBO)," in *Proc. ISCAS*, vol. 3, pp. 1054–1057, 1986.

[25] L. H. Goldstein and E. L. Thigpen, "SCOAP: Sandia controllability/observability analysis program," in *DAC* (E. B. H. Jr., ed.), pp. 190–196, ACM/IEEE, 1980.

[26] F. Brglez, "On testability of combinational networks," in *Proc. of International Symposium on Circuits and Systems*, pp. 221–225, 1984.

[27] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *DAC*, pp. 530–535, ACM, 2001.

[28] J. P. M. Silva and K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Trans. Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[29] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[30] M. Davis, G. Logemann, and D. W. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[31] R. Sebastiani, "Lazy Satisability Modulo Theories," *JSAT*, vol. 3, no. 3-4, pp. 141–224, 2007.

[32] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. TACAS*, TACAS'08/ETAPS'08, pp. 337–340, Springer-Verlag, 2008.

[33] B. Dutertre and L. D. Moura, "The Yices SMT solver," tech. rep., 2006.

[34] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The MathSAT 4 SMT Solver," in *CAV* (A. Gupta and S. Malik, eds.), vol. 5123 of *Lecture Notes in Computer Science*, pp. 299–303, Springer, 2008.

[35] C. Barrett, A. Stump, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)." `www.SMT-LIB.org`, 2010.

[36] G. Dantzig, "Programming of interdependent activities: II. Mathematical model," *Econometrica, Chicago*, vol. 17, pp. 200–211, 1949.

[37] N. Karmarkar, "A new polynomial-time algorithm for linear programming," in *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pp. 302–311, ACM, 1984.

[38] Gurobi Optimization, Inc., "Gurobi Optimizer Reference Manual," 2012.

[39] IBM Software, "IBM ILOG CPLEX Optimizer." http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

[40] J. P. Marques-Silva, "On computing minimum size prime implicants," in *in International Workshop on Logic Synthesis*, 1997.

[41] D. Hochbaum, "An optimal test compression procedure for combinational circuits," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, pp. 1294 –1299, oct 1996.

[42] K. R. Kantipudi and V. D. Agrawal, "On the size and generation of minimal N-detection tests," in *in Proc. 19th International Conf. VLSI Design*, pp. 425–430, 2006.

[43] M. A. Shukoor and V. D. Agrawal, "A Primal-Dual Solution to the Minimal Test Generation Problem," in *Proc. 12th VLSI Design and Test Symp.*, pp. 169 –179, 2008.

[44] P. Flores, H. Neto, and J. Marques Silva, "An exact solution to the minimum size test pattern problem," in *Proc. Intl Conf. Computer Design*, pp. 510 –515, oct 1998.

[45] M. A. Shukoor and V. D. Agrawal, "A two phase approach for minimal diagnostic test set generation," in *European Test Symposium*, pp. 115–120, IEEE Computer Society, 2009.

[46] T. Iwagaki and M. Kaneko, "On the derivation of a minimum test set in high quality transition testing," in *Test Workshop, 2009. LATW'09. 10th Latin American*, pp. 1–6, IEEE, 2009.

[47] F. Fallah, P. Ashar, and S. Devadas, "Functional vector generation for sequential HDL models under an observability-based code coverage metric," *IEEE Trans. VLSI Syst.*, vol. 10, no. 6, pp. 919–923, 2002.

[48] A. Sagahyroon, F. A. Aloul, and A. Sudnitson, "Using SAT-Based Techniques in Low Power State Assignment," *Journal of Circuits, Systems, and Computers*, vol. 20, no. 8, pp. 1605–1618, 2011.

[49] Z. Zeng, K. Talupuru, and M. Ciesielski, "Functional test generation based on word-level SAT," *Journal of Systems Architecture*, vol. 51, no. 8, pp. 488–511, 2005.

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

[51] S. Hellebrand, S. Tarnick, J. Rajski, and B. Courtois, "Generation of vector pattterns through reseeding of multiple-polynomial linear feedback shift registers," in *Proc. Intl Test Conf.*, p. 120, sep 1992.

[52] S. Hellebrand, J. Rajski, S. Tarnick, S. Venkataraman, and B. Courtois, "Built-in test for circuits with scan based on reseeding of multiple-polynomial linear feedback shift registers," *IEEE Trans. Computers*, vol. 44, pp. 223 –233, feb 1995.

[53] S. Hellebrand, B. Reeb, S. Tarnick, and H.-J. Wunderlich, "Pattern generation for a deterministic BIST scheme," in *Proc. IEEE Intl Conf. Computer-Aided Design*, pp. 88 –94, nov 1995.

[54] A. Al-Yamani and E. McCluskey, "BIST-guided ATPG," in *Proc. Intl Symp. Quality of Electronic Design*, pp. 244 – 249, march 2005.

[55] A. Al-Yamani, S. Mitra, and E. McCluskey, "BIST reseeding with very few seeds," in *Proc. VLSI Test Symp.*, pp. 69 – 74, april-1 may 2003.

[56] C. Krishna, A. Jas, and N. Touba, "Test vector encoding using partial LFSR reseeding," in *Proc. Intl Test Conf.*, pp. 885 –893, 2001.

[57] H.-J. Wunderlich and G. Kiefer, "Bit-flipping BIST," in *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conf. on*, pp. 337 –343, nov 1996.

[58] N. Touba and E. McCluskey, "Altering a pseudo-random bit sequence for scan-based BIST," in *Proc. Intl Test Conf.*, pp. 167 –175, oct 1996.

[59] N. Touba and E. McCluskey, "Synthesis of mapping logic for generating transformed pseudo-random patterns for BIST," in *Proc. Intl Test Conf.*, pp. 674 –682, oct 1995.

[60] M. Kamal, M. Jelodar, and S. Hessabi, "GABIST: A New Methodology to Find near Optimal LFSR for BIST Structure," in *Electronics, Circuits and Systems, 2007. ICECS 2007. 14th IEEE International Conf. on*, pp. 1107 –1110, dec. 2007.

[61] E. Aleksejev, A. Jutman, and R. Ubar, "LFSR Polynomial and Seed Selection Using Genetic Algorithm," in *Baltic Electronics Conf., 2006 International*, pp. 1 –4, oct. 2006.

[62] M. Lempel, S. Gupta, and M. Breuer, "Test embedding with discrete logarithms," *IEEE Trans. CAD*, vol. 14, pp. 554 –566, may 1995.

[63] S. Prabhu, M. Hsiao, L. Lingappan, and V. Gangaram, "A Novel SMT-Based Technique for LFSR Reseeding," in *Proc. VLSI Design Conf.*, pp. 394 –399, jan. 2012.

[64] J. Hayes and A. Friedman, "Test point placement to simplify fault detection," *Computers, IEEE Transactions on*, vol. C-23, pp. 727 – 735, july 1974.

[65] B. Krishnamurthy, "A dynamic programming approach to the test point insertion problem," in *DAC*, pp. 695–705, 1987.

[66] A. J. Briers and K. A. E. Totton, "Random pattern testability by fast fault simulation," in *ITC*, pp. 274–281, IEEE Computer Society, 1986.

[67] D. Brand and V. S. Iyengar, "Synthesis of pseudo-random pattern testable designs," in *ITC*, pp. 501–508, IEEE Computer Society, 1989.

[68] Y. Savaria, M. Youssef, B. Kaminska, and M. Koudil, "Automatic test point insertion for pseudo-random testing," in *Circuits and Systems, 1991., IEEE International Sympoisum on*, pp. 1960 –1963 vol.4, jun 1991.

[69] B. Seiss, P. Trouborst, and M. Schulz, "Test point insertion for scan-based BIST," in *Proc. of European Test Conference*, pp. 253–262, 1991.

[70] H. Tsai, C. Lin, S. Bhawmik, and K. Cheng, "A hybrid algorithm for test point selection for scan-based BIST," in *Proceedings of the 34th annual Design Automation Conference*, pp. 478–483, ACM, 1997.

[71] N. Tamarapalli and J. Rajski, "Constructive Multi-Phase Test Point Insertion for Scan-Based BIST," in *ITC*, pp. 649–658, IEEE Computer Society, 1996.

[72] N. Touba and E. McCluskey, "Test point insertion based on path tracing," in *Proc. VLSI Test Symp.*, pp. 2 –8, apr-1 may 1996.

[73] Y. Fang and A. Albicki, "Efficient testability enhancement for combinational circuit," in *ICCD*, pp. 168–179, IEEE Computer Society, 1995.

[74] J. Silva and K. Sakallah, "Robust search algorithms for test pattern generation," in *Proc. Intl Sympo. Fault-Tolerant Computing*, pp. 152 –161, jun 1997.

[75] S. B. Akers, C. Joseph, and B. Krishnamurthy, "On the Role of Independent Fault Sets in the Generation of Minimal Test Sets," in *Proc. International Test Conf.*, pp. 1100 –1107, 1987.

[76] M. Živkovic, "A table of primitive binary polynomials," *Math. Comput.*, vol. 62, pp. 385–386, Jan. 1994.