McGill University
Department of Electrical Engineering
Montréal, Québec, Canada

Compression and Decompression of Test Data for Scan Based Designs

Nadime Zacharia
B.Eng. (McGill) 1994

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of
Master of Engineering

December 1996

0-612-44048-6

Abstract

Traditional methods to test integrated circuits (ICs) require enormous amount of memory, which make them increasingly expensive and unattractive. This thesis addresses this issue for scan-based designs by proposing a method to compress and decompress input test patterns. By storing the test patterns in a compressed format, the amount of memory required to test ICs can be reduced to manageable levels. The thesis describes the compression and decompression scheme in details. The proposed method relies on the insertion of a decompression unit on the chip. During test application, the patterns are decompressed by the decompression unit as they are applied. Hence, decompression is done on-the-fly in hardware and does not slow down test application.

The design of the decompression unit is treated in depth and a design is proposed that minimizes the amount of extra hardware required. In fact, the design of the decompression unit uses flip-flops already on the chip: it is implemented without inserting any additional flip-flops.

The proposed scheme is applied in two different contexts: (1) in (external) deterministic-stored testing, to reduce the memory requirements imposed on the test equipment; and (2) in built-in self test, to design a test pattern generator capable of generating deterministic patterns with modest area and memory requirements.

Experimental results are provided for the largest ISCAS'89 benchmarks. All of these results point to show that the proposed technique greatly reduces the amount of test data while requiring little area overhead. Compression factors of more than 20 are reported for some circuits.

Résumé

Les méthodes de génération de vecteurs de test pré-déterminés requièrent des quantités impressionnantes de mémoire, ce qui les rendent de plus en plus impopulaires. Cette thèse propose une nouvelle méthode de génération de vecteurs pré-déterminés et vise tout particulièrement les circuits numériques avec une ou plusieurs chaînes de balayage. La technique compresse les vecteurs avant de les stocker, ce qui minimise la quantité de mémoire requise. Ils sont, par la suite, générés à l'aide d'une unité de décompression incorporée sur la puce.

La technique utilisée pour compresser et décompresser les vecteurs de test est décrite en détailles, de même que toute la théorie de base nécessaire à sa réalisation. De plus, la conception de l'unité de décompression est traitée en profondeur. Il est démontré que cette unité peut être réalisée avec quelques portes logiques, sans nécessiter aucune bascule.

La méthode proposée dans cette thèse peut être appliquée dans deux cas différents: (1) pour réduire de façon significative la quantité de mémoire qui doit être stockée par les équipements de test; et (2) pour la conception d'une unité de génération de vecteurs de test autonome qui peut émettre des vecteurs pré-déterminés sans nécessiter des quantités énormes de mémoire et de silicone. Cette dernière application est particulièrement importante pour les circuits avec fonction de vérification autonome et intégrée.

De nombreuses expérience ont été faite avec les circuits ISCAS89. Les résultats tendent à démontrer que la nouvelle méthode proposée dans cette thèse peut émettre des vecteurs pré-déterminés sans infliger des quantités excessives de données, et sans utiliser trop de silicone.

Acknowledgements

I would like to acknowledge Dr. Rajski and Dr. Tyszer for their guidance throughout the course of my studies. Their vision, hard work and their enthusiasm greatly contributed to this work. I will benefit from their advice, and their teaching for the rest of my career.

During my stay in the MACS laboratory, I deeply appreciated the friendship of the student members and staff. I would like to say hello to Mark Kassab, Nagesh Tamarapalli, Nilanjan Mukherjee, Victor Zia, Kasia Radecka, Vincent Leung, Jacek Slaboszawicz, and Alain Magloire, to name a few. They all added to my stay in the MACS laboratory. I wish them luck and happiness.

The experiments in the thesis were possible thanks to the help of many people. I am grateful to Robert C. Aitken and Pieter Trouborst for providing samples of test data for some industrial circuits. I also would like to thank John. A. Waicukauski for adapting his test generation programs to the methods proposed in the thesis.

On a more personal level, I would like to thank my parents, Abdallah and Helene, and my girlfriend Martine, for their encouragement and love all throughout my studies. I also would like to thank my brother Karim and my sister Nadia.

This work would not have been possible without monetary support. I was very privileged to receive several awards and bursaries throughout my studies. In particular, I acknowledge receiving two summer research bursaries from NSERC and the NSERC award (master level).

Table of Contents

List of Figures

List of Tables

Chapter 1 *Introduction*

Since their inception, digital integrated circuits (ICs) have undergone a tremendous rise in complexity and use. In the past twenty years, the number of transistors that can be fabricated on one chip has gone from a few dozens (small-scale integration or SSI), to few millions (very-large-scale integration or VLSI). During the same period of time, the functionality-to-price ratio has continuously increased which lead to a rapid and steady growth of the IC industry.

The level of quality of digital integrated circuits has a direct impact on the reliability and quality of electronic products. A large number of consumer goods, like computers, cellular phones, and digital radios comprise one or more digital ICs. If the ICs fail to operate normally, these goods cease to function and have to be replaced or repaired. Digital ICs are also part of embedded systems used as controllers in various transportation vehicles, like cars and planes. For these applications, failures of one or more ICs may be catastrophic and lead to fatal accidents.

Despite meticulous care, digital ICs can fail to operate properly. Some will fail because they were not designed properly. These chips are said to contain design or functional errors. Others will fail because of fabrication defects. These defects occur because the fabrication process of digital chips is not perfect and a fraction of them contain "imperfections" that may impact their behavior. Finally, chips, like all other physical device, will eventually fail due to their limited lifespan.

In order to guarantee high levels of quality and reliability of goods containing ICs, the ICs have to be tested for potential failures. There are many types of test depending on the type of errors or defects that are targeted. Functional tests check for functional errors. Production or manufacturing tests check for manufacturing defects. Maintenance tests check for wear-out in the field.

The general topic of this thesis is *production testing*. The goal of production testing is to identify badly manufactured circuits among the total number of chips produced. A perfect test procedure separates badly manufactured chips from good ones in a cost effective manner. However, ideal procedures do not exist and a fraction of ICs shipped to users contain manufacturing defects. This fraction is called the *defect level* and is often used to assess the quality of a test. Requirements for defect levels are usually measured in defective parts per million (dppm).

Currently, testing is predominantly achieved by applying deterministic-stored patterns by means of an external tester (Fig. 1.1). The test stimuli and expected responses are stored in the memory unit of the tester. During testing, the test stimuli are applied to the chip under test and the responses are collected and compared with the expected ones. If the responses of a chip are as expected, it is tested as "good" and shipped to users. Otherwise, it is discarded or subsequent experiments are conducted to identify why it failed (diagnosis).

However, several factors render this approach very unattractive for high density circuits. The memory requirements associated with the input test patterns are huge and may be beyond the capacity of test equipment[1]. The number of input/output (I/O) pins that can be driven by a tester is often limited by cost considerations. The corresponding limited channel capacity between the tester and the chip under test may impose long test application time. Last but not least, the computing resources required to generate the test (i. e..

1. In 1990, IBM testers for ASICs had only 64Mbytes of buffer memory [11].

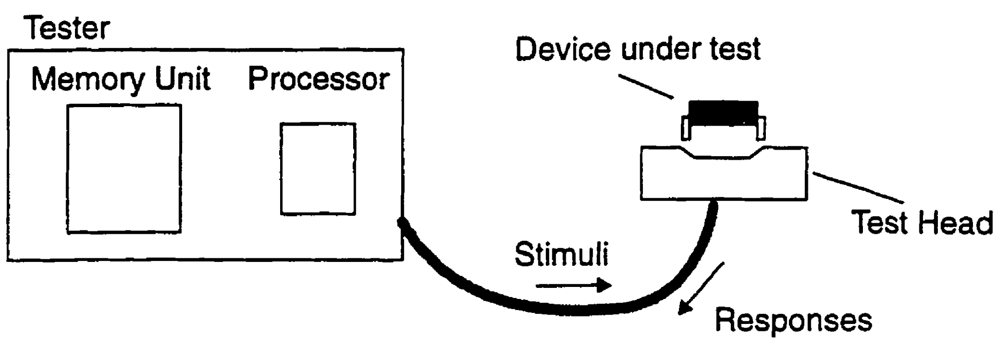


Figure 1.1: Deterministic-stored pattern testing with external tester

calculating what patterns to apply) are prohibitive.

Consequently, testing has become one of the most costly steps in the fabrication of an integrated circuit. Several surveys of the ASIC and IC industries have revealed that testing is in fact the most expensive step [1]. Furthermore, in some companies like Intel, the test equipment represents the **largest** capital investment of the whole company [47].

In order to reduce test costs, various *design-for-testability* (DFT) methodologies have been proposed. DFT methodologies are design rules and design techniques aimed at facilitating production testing. They are based on the premise that by considering testing at the design level, it is possible to produce circuits that are easier to test once manufactured.

One of the first DFT methodology that was proposed, and perhaps the most popular. is the *scan design methodology.* Scan methodology gives a mechanism to control and observe the state of a sequential circuit during testing. The problem of testing a sequential circuit is reduced to the problem of testing a combinational circuit, which tremendously simplifies the problem. Consequently, scan design methodology is extensively used to enhance the testability of sequential circuits. This thesis focuses exclusively on circuits with one or more scan chains. These circuits will be referred to as ***scan-based designs*** throughout the thesis.

Another key DFT methodology that is often used in conjunction with scan is *built-in self test*. Built-in self test adds test pattern generation and test response analysis as extra functions to be performed by the circuit. Hence, a circuit with built-in self test capabilities has extra functionality to test itself, i. e., apply its own test stimuli and verify that it functioned correctly.

Despite the use of DFT techniques like scan, the amount of test data required to achieve high test quality is still enormous. Table 1.1 shows the characteristics of a CMOS chip used as examples in [11, 28]. In order to achieve high test quality, 27K deterministic patterns have to be stored and applied. Each input pattern consists of 16K bits. Hence, 54Mbytes of memory are required to store the input patterns. A board with ten chips with similar characteristics would require more than 0.5 Gbytes of test data.

Number of gates	300K
Number of I/Os	512
Number of deterministic patterns	27K
Memory requirements of one input pattern	16K bits
Memory requirements of deterministic-stored testing	54Mbytes

Table 1.1: Characteristics of circuit used in example

Worse yet, the volume of test data tends to grow more than linearly with circuit size [11, 28]. The volume of test data is proportional to the product of the number of deterministic patterns and the size of each pattern. Whereas the former grows less than linearly with circuit size, the latter tend to grow linearly. The product of the two parameters, therefore, grows more than linearly with circuit size. According to this observation, a chip with similar characteristic as the one used in Table 1.1 but composed of 600K gates would require more than 108Mbytes of storage for the deterministic patterns. The rapid growth of test data with circuit size also implies that even if deterministic-stored pattern testing is feasible for the present generation of ICs, it may not be feasible for the next generations.

Reduction of memory requirements of inputs patterns may have a major impact of the cost of testing. Storing and managing these patterns is very costly because memory is one of the most expensive and heavily used resource during testing. Furthermore, the capacity of channels between components is often limited, which delays transfers of test data. By reducing the volume of test data, all of these costs and limitations could be reduced. For example, less expensive test equipment could be used, and the test procedure could be fasten.

Random pattern testing (with Signature Analysis) is a popular approach to reduce memory requirements. With this approach, a chip is exercised with random (or pseudo-random) rather than stored-deterministic patterns. Since there exists many methods to generate random patterns with minuscule memory usage compared to deterministic patterns, the overall technique has the potential to reduce test data volumes. Furthermore, methods to generate random patterns can be implemented with reasonable silicon area. They are therefore very attractive for built-in self test applications for which the test pattern generator is to be incorporated on the chip.

However, even if randomly generated stimuli are applied, the memory requirements may still be significant for a high quality test. These requirements arise because deterministic patterns have to be used to complement the random patterns to increase the test quality to acceptable levels. The required memory may represent 70% or more of the total memory requirements of deterministic-stored patterns [11, 28]. For the example of Table 1.1, 70% of the deterministic data translates to 38Mbytes of memory. A board composed of ten chips with similar characteristics would require 380Mbytes of test data. Clearly, using random patterns is not enough to reduce the memory requirements to acceptable levels.

The same observations can be drawn from circuits with BIST. One of the objectives of any BIST scheme is to incorporate in the circuit a test pattern generator that can achieve high test quality at modest costs. The costs of a BIST generator include the additional silicon required to implement the generator and the memory requirements to store the test data. In most cases, the memory requirements are too large for practical implementations. A BIST generator for the circuit of Table 1.1 would need at least 38Mbytes of memory, just to store the deterministic test patterns.

1.1 Overview of Thesis

This thesis proposes a new scheme to reduce the amount of test data for scan-based designs, while maintaining high test quality and low cost. The reduction in data volume is accomplished by compressing the deterministic patterns prior to storing them in memory. Later, during test application, the patterns are re-generated by decompressing them using a special on-chip decompression unit. The thesis demonstrates how to implement the decompression unit with minimal silicon overhead. Most of the hardware required to implement the unit is re-used from modules usually incorporated on a wide range of designs.

The scheme can be applied for circuits with BIST, for which the patterns are generated on the chip or for circuits without BIST, for which the patterns are generated by an external tester. In both cases, the test costs are greatly lowered by reducing the test data volumes. Extensive experiments with the largest ISCAS'89 circuits have been performed. The results of these experiments demonstrate that the proposed scheme greatly reduces the amount of test data while maintaining low costs and high test quality.

The thesis is structured as follows; the next Chapter reviews basic methods and concepts to test digital circuits at the logic level. Then, Chapter 3 introduces scan-based designs and gives further motivations for this work. Chapter 4 presents the mathematical background and mechanism used to compress and decompress deterministic patterns. These concepts are used in Chapter 5 in the implementation of the scheme for scan-based designs with and without BIST features. Experimental results using the scheme are gathered in Chapter 6. Finally, Chapter 7 concludes. Note that the appendix contains some additional material pertaining to the thesis.

Chapter 2 *Basics of Logic Testing*

2.1 Introduction

An *n*-input, *m*-output digital circuit is a network of components that accepts *n* binary signals as inputs and produces *m* binary signals as outputs. A digital circuit is described at various levels of abstraction depending on the type of primitive components recognized. At the switch level, it is described using only on-off switches. At the logic level, the circuit is represented with logic gates and flip-flops. At the behavioral level, the circuit is described using more complex components, like registers, ALUs, and memory units.

There are two main classes of digital circuits. Those that do not have internal memory are called *combinational circuits*. At any time, their outputs depend only on the values on the primary inputs. *Sequential circuits*, in contrast, have internal memory or an internal state. Their outputs depends both on the present input and their current state.

Sequential circuits can be *synchronous* or *asynchronous* depending when state transitions occur. In synchronous circuits, state transitions are synchronized with a special global timing signal called a *clock*. Transitions in an asynchronous circuit are not synchronized with a global signal.

This chapter reviews some testing techniques for combinational and synchronous sequential circuits.

2.1.1 Huffman Model of Sequential Circuits

Sequential circuits can be partitioned into two main parts (Huffman model [20, 25]): a combinational part, and a memory part. The combinational part has two main inputs and two main outputs. Its inputs are the primary inputs of the circuit and the current state of the machine, while its outputs are the primary outputs of the circuit and the next state of the machine. The memory part stores a finite amount of information which is referred to as the *state* of the circuit. The state of the circuit changes only in synchronization with a clock signal. As the clock signal changes (rising-edge), the next state becomes the new current state.

Fig. 2.1 shows the structure of a sequential circuit. It is composed of a combinational part (C) and a memory unit (R). Since the memory part is implemented with a parallel register, it is often referred to as the *state register*.

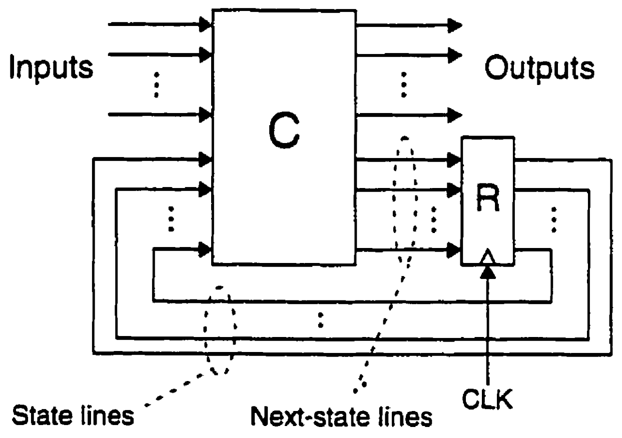


Figure 2.1: Huffman model of sequential circuits

2.1.2 Testing

Digital circuits can fail to operate properly either because they were not properly designed or because they developed physical defects due to fabrication errors or use. The number of design errors can be minimized by using CAD tools to simulate and verify the design. The occurrence of physical defects can be reduced but cannot be entirely eliminated by improving the fabrication process, improving the operating conditions of the circuit and using only reliable components.

Since the occurrence of physical defects cannot be completely eliminated, it is very important to test digital circuits to minimize the number of defective ICs on the market. A good measure of the quality of a test is the *defect level* [1]. The defect level is defined as the ratio of the number of defective ICs over the total number of ICs shipped. Obviously, the better the quality of a test, the lower the defect level for a given product manufactured to a given yield.

The testing process consists of three main steps: (1) test generation, (2) application of test, and (3) evaluation of the results. Test generation produces a test sequence that will determine, if applied, whether the circuit is properly functioning. This step is usually done prior to manufacturing the circuit. The test sequence is then applied to the circuit during test application. Finally, the response is evaluated against the expected response of the circuit.

A digital integrated circuit is tested more than once over its lifetime to insure that it is working property. During production of a digital system, a digital IC is tested at each level, i.e., (1) immediately after fabrication as part of the wafer; (2) after it is cut from the wafer and installed in plastic or ceramic package; and (3) after it is mounted over a printed circuit board.

The costs of finding bad parts tend to increase by an order of magnitude when going to a upper level [8]. Hence, if it costs \$1 to detect a bad chip at the wafer level, the costs at the packaging and PCB levels can be expected to be \$10, and \$100, respectively.

2.2 Fault Models and Tests

There are many sources of physical failures for digital integrated circuits. During fabrication, some parts can be missing, defective or fabricated by mistake. The interconnections between parts can also pose problems. Some interconnections may be missing or be wrongly inserted. During use, abnormal conditions, like excessive heat, inadequate voltage and current supplies, can affect the operation of a circuit and cause failures. Even if all operating conditions are normal, integrated circuits, like all physical devices, will eventually fail due to their limited lifespan.

Physical failures are classified according to several properties. For example, one can distinguish between *permanent* and *intermittent* failures depending whether they affect the operation of the circuit permanently or not. *Latent* defects are defects that are not visible right after fabrication but have the potential to become active later on.

Digital systems are analyzed and tested at different levels to account for potential failures. Testing carried out at the electrical level is called *parametric testing* because it measures voltage waveforms, propagation delays, currents, and the other parameters of the circuit. *Logic testing*, on the other hand, carries out testing at the logic levels and is only concerned about the logic behavior of the circuit. This thesis is primarily concerned with logic testing.

In order to perform logic testing, a fault model has to be used to relate the impact of physical failures on the behavior of the circuit. Many different models have been proposed, including the *single stuck-at fault* [1, 15, 35], *multiple stuck-at fault* [42], *stuck-on*, *stuck-open* [49], *bridging fault* [45, 15, 35], *transition fault* [1], and *path delay* [30, 46] fault models.

2.2.1 Single Stuck-at and Other Fault Models

The *single-stuck-at fault* (SSF) model is by far the most popular fault model. It assumes that a faulty circuits has a single line stuck at 0 or 1. A line $\mathcal{L}$ is said to be stuck-at v if it has value v irrespective of the correct logical output of the gate driving the line.

The SSF model has several properties that are worth mentioning. A circuit with n lines can have $2n$ distinct single stuck faults. Hence, the maximum number of different fault to consider is a linear function of the number of lines in the circuit. The SSF model is independent of technology and can be used for CMOS, TTL, or ECL logic. More importantly, even though single stuck-at fault do not represent directly the behavior of most physical failures, good tests for them detect most physical failures.

The *multiple stuck-at fault* model assumes that a faulty circuits has one or more lines stuck-at. Hence, each line in the circuit has three modes of behavior. It can be stuck-at 0, stuck-at 1 or it can have its fault free behavior. Therefore, a circuit with n lines has $3^n - 1$ possible stuck-at fault combinations. This number is very large, even for moderately sized circuits, which makes this fault model difficult to use. The multiple stuck-at fault model is rarely used in practise because it has been shown that tests that target single stuck-at faults detect most multiple stuck-at faults [2].

Stuck-on and *stuck-open faults* closely model defects that affect CMOS transistors. Due to such defects, some transistors may be permanently conducting (stuck-on) or permanently cut-off (stuck-open). As opposed to the stuck-at fault model, these fault models are technology dependent since they are based on CMOS technology. Furthermore, faulty circuits that have stuck-open faults may have memory due to capacitance of lines and thus may require a sequence of test patterns to detect one fault.

A circuit with *bridge faults* has two or more of its lines wrongly connected together. As a result, the signal corresponding to the group of lines may have an unpredictable logic value when the individual lines have conflicting assignments. The *delay fault* model considers unacceptable delays along paths from input to output. This fault model is used to verify that a design meets its performance specifications.

The *transition fault* model also considers delays in a circuit. A faulty circuit has either a slow-to-rise or a slow-to-fall fault along a path from input to output. A slow-to-rise fault occurs when an output is slow to rise from 0 to 1 while a slow-to-fall fault occurs then an output is slow to fall from 1 to 0. Detection of a transition fault involves two patterns. One pattern to set the initial value on the path to the output, and one pattern to create the transition. Note that the timing between patterns is very important.

2.2.2 Testing for Faults

In order to determine whether a circuit contains a fault *f* or not, some patterns have to be applied. For combinational circuits, only one pattern is required if such pattern exists. For sequential circuits, however, a sequence of one or more patterns is necessary, if such a sequence exists. The remaining parts of this section concentrates on combinational circuits for simplicity.

A test vector *V covers* or detects a fault *F* if and only if applying *V* to the circuit produces a different output whether the fault is present or absent from the circuit. Determining the test vector that cover a fault is a process called test generation and is discussed in Section 2.3.

Faults are classified according to their detectability and distinguishability. A fault is *undetectable* or *redundant* if and only if there does not exist any vector that covers it. In contrast, detectable faults can be detected by at least one vector. A group of faults are *indistinguishable*, or *equivalent*, if they affect the external behavior of a circuit in the exact same way.

A *complete test set* is a set of test vectors that cover all the faults in the circuit. A complete test set is *minimum* if it is the complete test set with the fewest number of vectors.

2.3 Test Generation and Fault Simulation

Given a circuit and a set of detectable faults *F*, *test generation* is the process of finding a sequence of test patterns that covers all faults in *F*. Despite the fact that it is an NP-Complete problem, several algorithms have been proposed tu automate this process and are referred to as *automatic test pattern generation* (ATPG) algorithms. These algorithms are divided in two important classes: those that target combinational circuits and those that target sequential circuits. ATPG algorithms that target combinational circuits include the D algorithms [40], PODEM [17], FAN [16], and SOCRATES [43]. Those that target sequential circuits include EBT [31], BACK [14], and FASTEST [27].

For a circuit with *n* inputs, the search space of combinational circuits is of order 2^n for each fault. For sequential circuits, the additional dimension of time makes the search space even bigger. Even if the number of patterns required to detect the fault is known or limited time unrolling is used, the search space is still enormous. As a result, sequential test generation is rarely used for large circuits when detection of all faults is important.

Many practical methods for combinational circuits take the following approach. A fault *f* stuck-at *v* is selected for which a test pattern *T* is required. The fault is excited by forcing the line to be at value *v*-bar. Then, a path is traced from the fault site to one of the primary outputs. For each gate along the path, the signals at the gate input are assigned such that the error signal can propagate. Finally, all of these signals are justified working backward in the circuit, until the primary inputs are reached.

Fig. 2.2 shows an example in which line *B* is stuck-at 0. The fault is excited by the assignment *B*=1. Then, a path is traced to propagate the fault effect the output. The path is sensitized by assigning non-controlling values to each gate along the path. Hence, *C*=1, *D*=0, *h*=1. After that, the signals are justified until the primary inputs are reached. The only unjustified signal is *h*. To justify the assignment *h*=1, the input *A* must be equal to 1. Combining input assignments for propagation and justification the test vector: *ABCD*=1110 is generated.

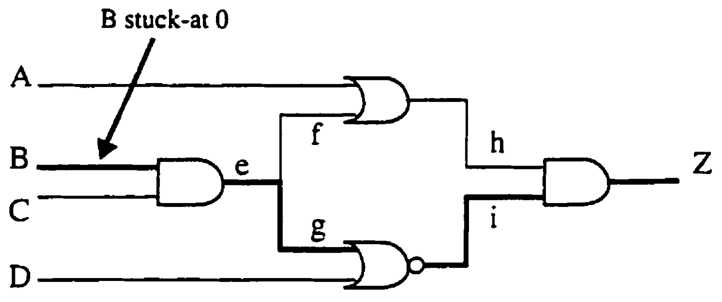


Figure 2.2: Test generation using path sensitization

During sensitization and justification, different kinds of signal assignments are made. Some are *necessary*, that is, they must be forced for a test to exists. Others are *arbitrary*, that is, alternative signal assignments could have been made.

During justification, it is possible to encounter contradictory requirements. To resolve such conflicts, it is necessary to go back to a previous arbitrary assignment and make an alternative decision. This process is called *backtracking*. If there are no arbitrary assignments, or there are no more alternative decisions to make, the fault is not testable.

In general, a test pattern generated by ATPG algorithms does not specify all the primary inputs. Some inputs are unspecified and may be either 0 or 1. For example, the pattern ABCD=11X0 detects the fault A stuck-at 0. In this pattern, C can be either 0 or 1. Hence, both patterns ABCD=1100 and ABCD=1110 detect A stuck-at 0.

Given a circuit, a fault list, and a set V of test patterns, *fault simulation* is the process of determining which faults are covered by the patterns in V. Fault simulation is often used to calculate the *fault coverage* of a set of patterns, i. e., the fraction of faults detected over the total number of potential faults in the circuit. Fault simulation is also used to identify undetected faults.

There exists many different methods to fault simulate a circuit. These methods are classified into four main categories: fault injection, deductive methods, concurrent methods, and critical path tracing methods.

Fault injection methods proceeds as follows. First, the fault free machine is simulated for a given pattern. Then, for each fault in the circuit, the fault effect is propagated forward in the circuit. The fault is detected if the fault effect reaches one of the primary outputs. Otherwise, it is not detected by the pattern. Once detected, a fault may be removed from the fault list and marked as detected to speed up fault simulation of the next patterns. This process is called *fault dropping*.

Fault injection comprises *parallel fault simulation* method [1] in which a computer word of w bits is associated with each line in the circuit. Each bit of the memory word corresponds to one faulty circuit. By using bit-wise computer instructions, w faulty circuits can be simulated in parallel.

Deductive fault simulation [7] maintains for each line in the circuit, the list of faults that propagate to the line. The algorithm first simulates the fault free circuit and then uses recursive rules to generate the list of detectable faults. The faults detected by a pattern are the faults contained in the list of detectable faults at the outputs of the circuit.

Concurrent fault simulation [1] maintains a list of faulty gates for each gate. The list contains the effect of each fault on the gate if it differs from the fault free state.

Critical path tracing [1] methods speed up fault simulation by considering only critical paths in the circuit. Once the fault free simulation is performed, critical paths are "traced" using the following rules recursively:

(1) The outputs are critical

(2) If the output of a gate is critical and only one input *j* has the controlling value, then signal *j* is critical.

(3) If the output of a gate is critical and all inputs have non-controlling value then all inputs of the gate are critical.

(4) Otherwise, no input is critical.

2.4 Test Pattern Generation

During test application, a number of patterns are applied to a circuit. There exist many different schemes to generate the input patterns. The ideal scheme covers all faults in the circuit by applying a small number of patterns, requires little memory to store test information and does not impose significant area overhead. Most schemes propose some kind of trade-off between the above parameters. The solutions include exhaustive, deterministic, pseudo-random and hybrid methods. Hybrid or mixed-mode methods combine two or more methods.

2.4.1 Exhaustive

Exhaustive testing methods [10] apply all possible input combinations to test a circuit. For a combinational circuit with n inputs, all 2^n possible input combinations are applied one by one. Since all input combinations are applied, all detectable faults are covered. Hence, exhaustive techniques obtain very high fault coverage. However, the number of patterns applied is an exponential function of the number of inputs. In most cases, the number of patterns is so high that it is impossible to apply them in a reasonable time. For example, a circuit with one hundred inputs would require 2^{100} patterns!

Exhaustive patterns can be generated by a simple counter without requiring any extra memory. Hence, exhaustive techniques achieve very high fault coverage and do not require a complex structure to generate the patterns, but they take very long test application time.

2.4.2 Deterministic

In *deterministic testing* [4, 11], the test patterns are pre-calculated off-line and stored in memory. They are calculated either manually or with the use of automatic test generation software. For combinational circuits most faults in the circuit can be targeted. Consequently, deterministic methods have potential for very high (or complete) coverage for combinational circuits.

During testing, the patterns are read one by one from the memory unit and applied to the circuit under test. The number of test patterns is relatively small and thus, the test application time is short. However, even if the number of test patterns is small, the memory required to store them may be very huge. Consequently, deterministic testing has to potential to achieve very high fault coverage (or complete fault coverage) with a relatively small number of patterns. However, the memory requirements may be very huge.

2.4.3 Pseudo-random

In pseudo-random testing [1, 8], a random generator is used to generate the patterns. Very simple pattern generators that do not require much area can be implemented in hardware. Popular generators include *linear feedback shift register* [18] and *linear cellular automata*

[24].

Pseudo-random pattern generators require minuscule amounts of memory to generate test patterns. However, since the generated patterns are random, the fault coverage may be a problem. Applying random patterns will cover most of the faults. These faults are called *easy-to-test*. However, a number of faults will not be covered even after a large number of random patterns are applied. These faults are called *hard-to-test* or *random pattern resistant* due to their "resistance" to random patterns. Most circuits have a large number of hard-to-test faults. For these circuits, it is impossible to achieve complete fault coverage in an acceptable number of patterns. Thus, pseudo-random techniques do not require memory to generate patterns but cannot achieve complete fault coverage in an acceptable number of test patterns.

2.4.4 Mixed-mode (Hybrid)

Mixed-mode or *hybrid techniques* [21-23, 28, 48, 52-53] combine the advantages of deterministic and pseudo-random patterns to offer many trade-offs in terms of memory requirements and test application time. Using this technique, a circuit is tested using a mixture of random and deterministic patterns. The random patterns are used to cover the easy-to-test faults while the deterministic patterns cover the remaining hard-to-test faults.

Mixed-mode techniques achieve complete fault coverage by storing deterministic patterns to supplement random patterns. In addition, mixed-mode techniques use schemes to compress the deterministic patterns because for most circuits, the amount of memory required to store the patterns explicitly is too large [21-23, 28, 48, 52-53]. Compression of the deterministic patterns is the topic of this thesis and an effective method for scan-based designs will be proposed in Chapter 5.

2.5 Test Response Analysis

The naive approach of verifying the response of the circuit under test against the fault-free response also leads to important memory usage. Using this approach, the response of the circuit is compared with the expected fault-free response for each pattern applied to the circuit. The fault-free response is either generated concurrently with the test or stored in memory. In both cases, this creates substantial memory requirements.

Several techniques have been proposed to facilitate response analysis by compacting the whole response of the circuit during testing into a single memory word called a *signature*. At the end of testing, the signature of the circuit is compared to the fault-free signature. There is no need to make a comparison for each pattern and no need to store the expected responses.

Since the complete response of the circuit is reduced into a single word, some information is loss which may lead to *aliasing* [8]. Aliasing occurs when a faulty response is compacted into the fault free signature. When this occurs, the defective chip that produced the faulty sequence is tested as being good which impacts negatively the quality of the test procedure. The probability of aliasing is a key parameter when comparing different compactors.

The most popular compaction technique is *Signature Analysis* [8], where a multiple input shift register (MISR) is used to compact the response into a signature. The analysis of MISR is beyond the scope of this thesis. However, as a rule of thumb, the aliasing probability of a k-bit MISR is approximately 2^{-k}. For a 32-bit MISR, the probability of aliasing is only 0.0000000002.

Other compaction techniques include *one's counting*, *transition counting*, and *one's complement addition*. In one's counting [8], the signature is calculated by counting the number of one's in the circuit response. In transition counting [8, 19], the number of 0-to-

1 and 1-to-0 transitions is counted. Compaction schemes based on one's complement addition [38] calculate the signature by successively adding the circuit's response using one's complement arithmetic. By using a *k*-bit adder, these schemes obtain a probability of aliasing approximately equal to 2^{-k}.

The fault-free signature is obtained by simulating the test patterns on a digital simulator. At the end of the simulation, the state of the signature register is recorded and is used as the faul-free signature. During the simulation, if a high impedance or an unknown value reaches the signature register, it will corrupt its current state and the fault-free signature will be unknown. Thus, compaction techniques can only by applied if the response of the circuit is strickly binary for every test input. For scan-based designs, this restriction means that all the memory elements in the scan chains must have known values during testing. Most designs do not abide by that restriction and signature analysis cannot be applied. However, some designs, those with built-in selft test, respect that restriction and signature analysis can be used to determine if the circuit is faulty or not.

2.6 Design for Testability

Testability is the set of characteristics that allows a circuit to be tested in a time and cost effective manner [20, 33]. Testability is difficult to quantify. However, the properties of easy-to-test designs are easily enumerated: (1) a test procedure detects a large portion of the potential faults in the circuit. (This proportion is called the *fault coverage*. Detection of all faults, or *complete fault coverage* is often a requirement of a test.) (2) test generation and application times are short, and (3) volume of test data is small.

If testability is not taken into account at the design stage, testing a digital circuit is likely to be a very costly and time consuming problem. Several factors would contribute to make testing a difficult problem. For complex designs, the pin/logic ratio is low which makes it extremely difficult to control and observe lines in the circuits. Also, the time required to generate tests is very long and many faults have to be abandoned by ATPG. Hence, complete fault coverage is difficult to obtain. Finally, the volume of test data is very huge and may cause memory problems during application of test sequence.

Design for testability are design-level techniques aimed at making a circuit easier to test [20, 33]. These techniques enhance testability mainly by improving the controllability and observability of lines in the design. To do so, most DFT techniques add additional built-in hardware which is measured in terms of *area overhead*, i. e., extra gates, flip-flops and pins. This hardware can be on the critical path of the circuit, which would impact the performance of the circuit. Thus, it is desirable to minimize the hardware overhead and performance degradation that may be imposed by DFT.

DFT techniques are divided into two main categories: *ad hoc* and *structured.*

2.6.1 Ad Hoc

Ad hoc techniques are lists of "do and don't" aimed at designers to avoid potential problems. Most of these rules attempt to enhance the controllability and observability of lines in the circuit. Examples of such rules are listed below:

Redundancy. Avoid redundancy that introduces undetectable faults.

Partition. Partition large hard-to-test sub-circuits into smaller, more easily tested ones.

Feedback. Allow feedback paths to be opened and closed during testing.

Test point. Add *test points* [44] to increase controllability and observability of selected lines (bottlenecks) in the circuit. A test point is either an additional fan-out to allow a line to be observed or additional circuitry to control the value of a line.

2.6.2 Structured (Scan and Built-In Self Test)

Ad hoc techniques lack generality and unity and tend to require many extra input and output pins. More systematic and structured approaches can be incorporated in the design process to guarantee ease of testability. One such approach is the scan design methodology which is discussed in Chapter 3. During testing, the state register of the circuit are configured into a shift register or *scan register*. The state is specified by shifting some data into the scan chain. Similarly, the state of the circuit is observed by shifting out the content of the state register.

Another key DFT methodology is *built-in self test* (BIST). Built-in self test methodology adds test pattern generation, test response compaction and test session control as key functions to be performed by the circuit. Hence, during testing, the circuit generates and applies its own test patterns and evaluate its responses [4, 5, 8]. In short, the circuit tests itself.

There are many reasons that motivate the use of built-in self test. In particular, built-in self test offers solutions to the following problems.

First, BIST reduces the needs for expensive test equipment. External testers are composed of very complex and high performance sub-systems, like processors and memory units that costs tremendous amount of money. Despite their high costs, because the technology progresses very quickly, testers tend to become out-dated very quickly and do not offer the capacity required to test state-of-the-art components. By having the system or chip test itself, the test hardware follows the pace of the technology.

Typically, BIST schemes have better access to the circuit under test than external testers. For instance, in order to test a chip, external testers have to pass through the primary inputs and outputs of the chip, which can be a bottleneck in some cases. By incorporating test functionality on the chip, the bottlenecks may be eliminated.

BIST offers a very attractive method to test proprietary (embedded) modules. Some companies, like ASIC vendors, sell libraries of components, including processor cores, controllers, and so on. Revealing information to allow customers to test the library components might allow them to reverse-engineer the actual designs. BIST allows for the components to be tested, while it frees the library vendor to release proprietary information.

More importantly, built-in self test offers a hierarchical solution to testing. It is applied at all levels of a design, and self-test functions are invoked at different levels. For instance, BIST can be applied at the system, board, or IC levels. Furthermore, a system self-test procedure may involve board self-test routines, which, in turns, involve the ICs self-test.

At the system level, self test is usually performed in software, while at the IC level it is performed in hardware. On an IC, self test is performed by three main units. The *test pattern generator* produces test patterns. The *test response analyzer* compacts the responses into a signature for later evaluation. Finally, the test sessions are controlled by a *test controller unit*. On an IC, these units are usually implemented in hardware to shorten test application time.

Fig. 2.3 shows the general architecture of built-in self test of an IC.

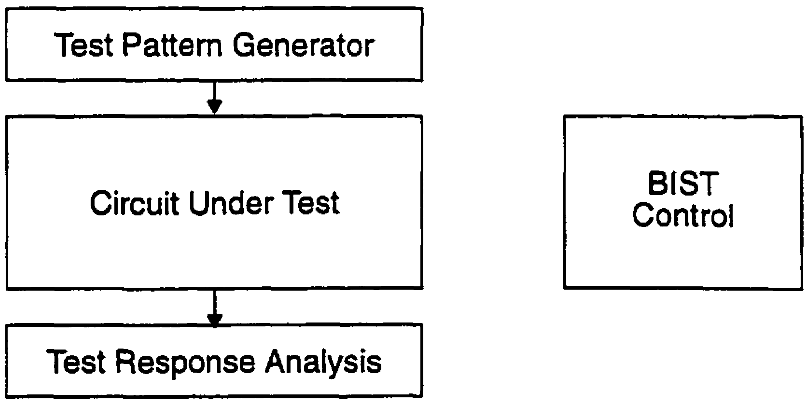


Figure 2.3: Built-In Self Test

Chapter 3 *Scan Design Methodology*

3.1 Introduction

Scan design methodology is the first structured design-for-test technique that was proposed. The first description of scan-based designs dates back to the 60's [51]. Since then, tremendous research has been conducted to develop the technique and make it one of the most mature. Scan design methodology is also one of the most popular DFT methods and is extensively used in the design of modern digital circuits. The most "publicized" version of scan is the Level Sensitive Scan Design (LSSD) technique developed by IBM [51]. Other scan related methods like boundary scan [IEEE/ANSI standard 1149.1-1990] are increasingly used by the ASIC and IC industries.

This Chapter introduces scan-based designs and the topology considered in this thesis. In particular, emphasis is put on multiple scan designs with Signature Analysis.

3.2 Full Scan Designs

The essence of scan design is to make the state of a sequential circuit completely controllable and observable. In order to do so, the parallel state register of the circuit is replaced by a parallel and shift register. During normal operations the register behaves as a parallel load register, while during testing, it behaves as a shift register. Implementing the state register implies adding extra gates to allow the flip-flops to be configured into a shift register during testing. The resulting configuration is called a *scan chain.*

Fig 3.1 shows the modified Huffman model in which the state register is replaced by a parallel-and-shift register. Three new pins are added: *scan_in*, *scan_out* and *scan_enable*. *Scan_enable* determines whether R is operating as a parallel register or a shift register. When operating as a parallel register, R loads data from the next state lines. When operating as a shift register, R shifts data serially, taking *scan_in* as input and *scan_out* as output. Using the extra pins, the state of the circuit is specified and observed by a test procedure.

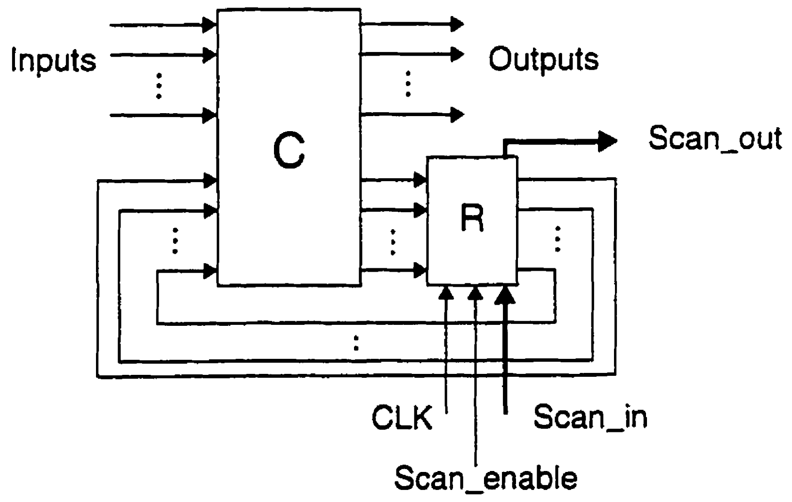


Figure 3.1: Sequential circuit with a scan chain

3.2.1 Modes of Operations

The resulting circuit has two modes of operations: Normal and Testing.

Normal mode. In normal mode, the circuit is operating its normal function. The state register behaves as a parallel register that stores the current state and loads at each cycle the next state.

Test mode. In test mode, the state register is configured into a shift register. Inputs *scan_in* and *scan_out* are used to shift data into and out of the register.

3.2.2 Test Pattern

In order to apply one test pattern to a scan-based design, a number of events have to take place. First, the state of the circuit and its inputs must be forced to the specified values. Then, the state of the circuit and its outputs have to be observed. Controlling and observing the state of the circuit is done by shifting data serially into the circuit.

The application of one pattern consists of four distinct events [20, 33]:

(1) Shift data into the scan chain/ receive data from the scan_out pin. For a circuit with a scan chain composed of *n* flip-flops, *n* clock cycles are required by this step.

(2) Force values on the primary inputs

(3) Measure values on the primary outputs

(4) Apply a clock transition

The time required to apply one test pattern is dominated by the first step, which requires *n* clocks for a scan chain composed of *n* flip-flops. Because the number of flip-flops in a circuit is much greater than the number of inputs and outputs, the data applied to the circuit is composed principally of the sequence loaded into the scan chain and the expected sequence to be loaded out from the scan chain. Since each pattern involves loading the scan chain, this method of applying patterns to a circuit is often labeled *test-per-scan*.

3.2.3 Test Generation

One of the most important benefits of scan is that test generation is reduced from sequential to combinational. This represents a major reduction in computational complexity as the dimension of generation time is eliminated from the problem. Furthermore, test generation for scan designs is reduced to test generation for combinational circuit since the state lines can be controlled and observed. Each state line can be seen as a *pseudo-input* of the circuit and each next state line as a *pseudo-output*.

Tests for scan designs have the potential to achieve very high fault coverage as most faults can be targeted by test generation using combinational ATPG. Methods based on sequential test pattern generation tend to achieve lower fault coverage because many faults have to be abandoned due to the computational complexity of sequential ATPG.

3.2.4 Trade-offs of Scan Designs

As was seen, scan design methodology trades area, performance and test application time to enhance fault coverage and reduce test generation time. In effect, multiplexers, additional primary inputs/outputs pins have to be inserted to implement the scan, which may represent more than 15% of addition hardware [33]. Test vectors have to be shifted serially in and out of the scan chain, which tends to give long test application time. In order to implement the scan chain, multiplexers have to be inserted in front of each flip-flop. Some of these multiplexers are likely to be in the critical path of the circuit and will therefore influence the performance of the circuit. Furthermore, it is very difficult to test the circuit at operational speed due to the slow loading and unloading of patterns. Finally, scan design methodology imposes many rules to designers which makes the design effort more laborious.

Despite these drawbacks, scan is often the only method to achieve complete fault coverage of sequential circuits. Other methods which rely on sequential test pattern generations rarely obtain complete fault coverage, as a large number of faults are dropped because of the large search space.

Still, some methods have been proposed to reduce the penalty of scan while keeping most of the advantages. The *partial-scan method* uses a subset of the flip-flops in the circuit to build the scan chain. During testing, the circuit behaves as a sequential circuit and a sequence of scan-patterns may have to be applied to cover a fault. However, by carefully choosing the flip-flops in the scan chain, it is possible to attain high fault coverage. The *multiple scan method* uses more than one scan path to shorten the test application time. It is considered next.

3.3 Multiple Scan Chains

One of the disadvantages of scan design methodology is the long time it takes to load data in and out of the scan chain. Large designs have thousands of flip-flops. Loading them serially, for each test vector would impose unacceptable delays. One method to reduce test application time is to break the scan chain into many parallel scan chains. Instead of having one state register, a circuit has *n* parallel-shift state registers. The corresponding scan chains have their own *scan_in* and *scan_out* line but are controlled by the same *scan_enable* line and the same clock signal.

Test patterns for a circuit with *n* scan chains are loaded *n*-bits at a time. Therefore, the test application time is reduced by a factor of *n* compared to a similar single scan design. For example, a circuit with a single scan chain with 1600 flip-flops would require 1600 clocks per pattern, whereas, a circuit with 16 scan chains, each composed of 100 flip-flops require 100 clocks per pattern.

Most scan-based designs have multiple scan chains to insure that the test application time is reasonable.

3.4 Boundary Scan

Scan design principles can be applied to any signals, not only to the state lines. In particular, they can be applied to the input and output ports to form a *boundary scan chain* [32]. This technique is particularly useful to test interconnections between ICs on a printed circuit board (PCB) and to access each IC with very few lines.

Design with boundary scan attach a special scan cell to each input/output port and link them serially in a scan chain. On a printed circuit board with boundary scan, the scan chains of each IC are linked together (Fig. 3.2). By setting special control signals, the pins of each ICs are observed and controlled by shifting data in and/or out of the boundary scan. Also, access to the internal scan chains can be gained to test each IC individually.

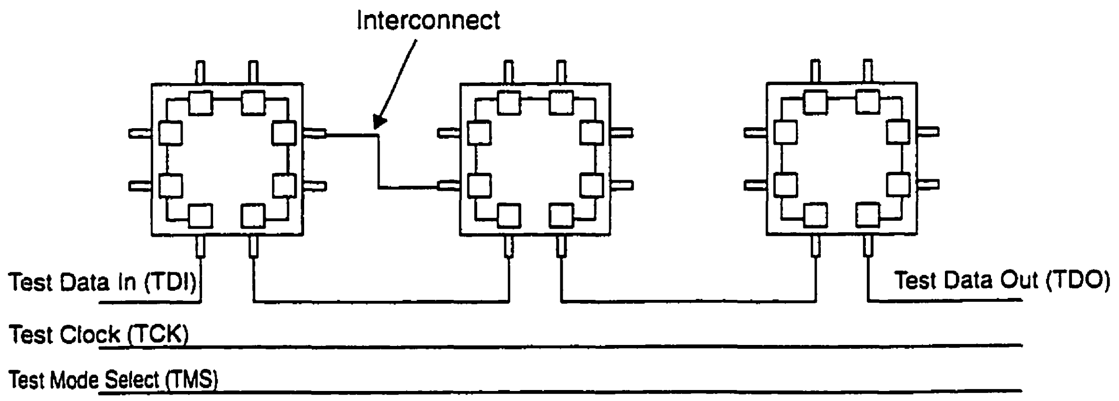


Figure 3.2: Boundary Scan

A standard implementation of boundary scan has been adopted by many circuit manufacturers (IEEE/ANSI standard 1149.1-1990). The standard specifies not only the design of the scan cell but also the boundary test protocol. An IC with boundary scan contains a test access port (TAP) controller and provides four extra pins: TCK(test clock), TMS(test mode select), TDI(test data in), and TDO(test data out).

The TAP controller handles the test protocol and should support the following modes:

(1) External: In this mode, the interconnects between ICs on a board are tested.

(2) Internal: This mode allows for the modules on the chip to be tested. (This mode is not a strict requirement bit is usually implemented)

(3) Bypass: In this mode, the boundary scan of the IC is bypassed.

(4) Sample: The values on the primary inputs and outputs are *sampled* into the boundary scan.

(5) Built-in self test: If the chip has built-in self test capabilities, this mode instructs the chip to test itself.

3.5 Test Pattern Generation for Scan

There are many approaches to generate patterns for scan-based designs. As was seen in Chapter 2, these methods are classified as random, deterministic, exhaustive, and hybrid.

3.5.1 Classical Approaches

As was seen in chapter 1, the most prevalent method for scan-based design is deterministic-stored pattern testing. Using this method, the test patterns that achieve complete fault coverage are stored in memory and are applied successively by reading the content of the memory. However, the large amount of memory required makes this approach highly unattractive. For each pattern, the sequence to be shifted in the scan chain has to be stored as well as the expected response to be shifted out.

This section reviews two major techniques that were proposed to reduce the amount of test data. Both techniques are based on mixed-mode test generation: applying random patterns to cover the majority of faults and applying deterministic patterns to cover the remaining faults. The techniques are *weighted random pattern testing* and *reseeding of multiple-polynomial linear feedback shift register.*

3.5.2 Weighted Random Pattern Testing

Often, hard-to-test faults are difficult to cover because they require test patterns with a large proportion of 0s or 1s. Such patterns are rarely generated by a random generator for which the probability of generating a binary 1 is 0.5. For example, a 20-input AND gate requires all its input to be high to detect the output stuck-at 0. If the inputs of the AND gates are randomly generated, this input pattern will occur with probability 2^{-20}.

Weighted random pattern testing [11, 50] uses a special circuitry to bias the bit probability of a random generator to increase the likelihood of generating patterns that will cover hard-to-test faults. Essentially, deterministic patterns are encoded as weight information, i. e., the proportion of 1s in the generated sequence. This weight information is organized in *weight sets*. Each weigth set contains weight data for each input of the circuit, including the pseudo-inputs.

Fig. 3.3 shows an example of a circuit that produces sequences with weights 0.5, 0.25, 0.125, 0.0625, 0.75, 0.875 and 0.9378. Each line in the circuit is labeled with the weight of the corresponding signal. The circuit uses the well-know property of AND gates, i. e., the weight at the output of an AND gate is the product of the weight at its inputs. A multiplexer is used to select which biased sequence to use. The XOR gate at the output is a programmable inverter. An inverter generates a sequence with weight $1\text{-}w$ if its input sequence has weight w.

The circuit uses three lines to select the weight of the sequence to be generated. In

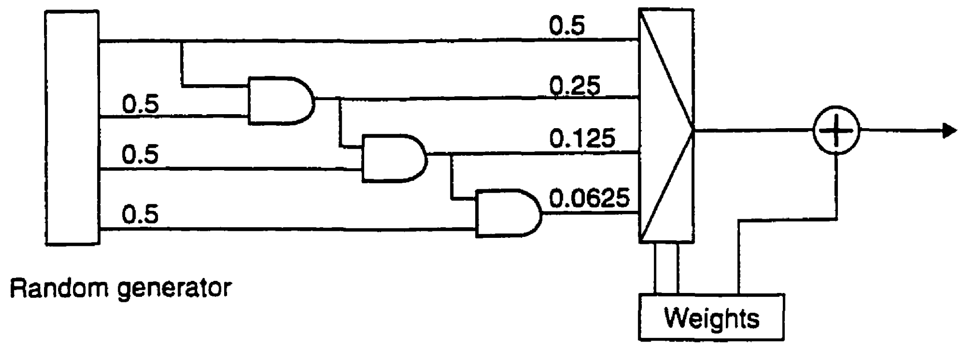


Figure 3.3: Weighted sequence generator

most practical implementations, four bits are used. In [11], it was shown that weighted random testing greatly reduces the amount of memory required to store deterministic patterns. However, weighted random testing has several drawbacks

(1) For a circuit with n scan chains, a pattern generator with $4n$ flip-flops is required to generate the different weighted sequences. Hence, very large generators are required. For applications were the generator is to be placed on the chip (for instance, built-in self test), the overhead is enormous and preclusive.

(2) Each weight set is composed of four bits for each input or pseudo-input. Large designs contain thousands of inputs and pseudo-inputs and need hundreds of weight sets. Consequently, even though weighted random pattern testing reduces the test data volumes, it may still impose excessive memory requirements.

(3) The number of patterns required to obtain complete fault coverage is an order of magnitude greater than for deterministic testing [21]. This lengthens the test application time dramatically which may make it costly for production testing.

3.5.3 Reseeding of Multiple-Polynomial Shift Register

In general, test pattern generation algorithms targeting a set of faults produce test cubes[1] with very few specified bits. Deterministic cubes targeting hard-to-test faults, for instance, often have less than 30% of their bits specified. Furthermore, the number of specified bits in test cubes varies significantly. Some test cubes will have 30% of their bits specified while others will have fewer than 5% bits specified.

Fig. 3.4 shows typical distributions of the number of specified bits in deterministic test cubes [52]. In the first graph, the number of positions in each cube is 436, while the number of specified positions varies from 4 to 67. Thus, not more than 15% of the positions are specified in the test cubes. Similarly, in the second graph, the number of positions in each cube is 2522. The number of specified positions varies from 5 to 137. In all test cubes, less than 5% of the bits are specified.

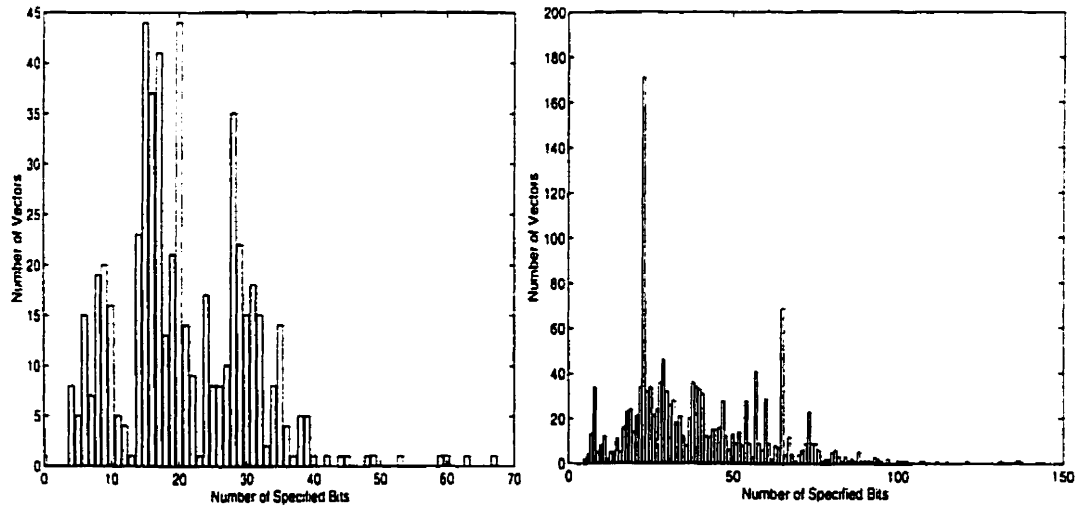


Figure 3.4: Typical distributions of number of specified bits in deterministic test cubes.

1. In this thesis. cubes are "seen" as sequences of 0s. 1s and x(don't cares). A position that is either 0 or 1 is said to be specified,while a position that is x is said to be unspecified. For example. the sequence (0. 1. x) is a cube with two specified positions. See chapter 4 for more information on cubes.

Very few methods have been proposed to compress and decompress test cubes by exploiting the fact that, often, they have very few specified positions. [28] proposed a very attractive approach which encodes deterministic patterns as seeds of a *linear feedback shift register* or LFSR. Each deterministic patterns is generated by the LFSR in two main steps. First, the LFSR is initialized with the corresponding seed. Then, it is clocked for a fixed number of cycles to fill the scan chains.

It is much more economical to store LFSR seeds rather than the corresponding deterministic patterns: an LFSR seed requires much less memory than a deterministic pattern. Furthermore, the seeds are calculated by solving systems of linear equations (modulo 2), which is a computationally simple operation. Hence, the conversion between seeds and deterministic patterns is performed rapidly and cheaply.

The approach assumes that there is an LFSR on the chip to generate the deterministic patterns. For some circuits, the LFSR has to be inserted at the expense of many flip-flops. For other circuits, like those with built-in self test, an LFSR is already on the chip to generate random patterns (see Section 3.6.1). The LFSR can be re-used to generate deterministic patterns. In essence, the LFSR is used as a mixed-mode generator of patterns. It is cycled in autonomous mode to generate pseudo-random patterns to cover the easy-to-test faults. Then, it is re-loaded with pre-computed seeds to generate deterministic patterns targeting the remaining hard-to-test faults.

The technique was improved in [21] and [22] by encoding groups of cubes rather than each cube separately. The techniques exploit the fact that the LFSR sequences are independent of the length of the test cubes. Thus, it is possible to encode a group of test cubes rather than one cube at a time. By using clever concatenation techniques, the group of cubes are encoded more efficiently than if they were encoded separately [22]. As a result, the compression effectiveness was significantly enhanced. On the average, a test cube with s specified positions is encoded with an s-bit word which yields high compression ratios.

The decompression structure is a programmable LFSR, called a *multiple-polynomial* LFSR (MP-LFSR) with 16 possible feedback configurations. Through special control lines, its feedback network is programmed to implement one out of the 16 possible configurations. Fig. 3.5 shows the general architecture of MP-LFSR reseeding as proposed in [21-23, 48]. The MP-LFSR is used to generate the deterministic patterns. Each group of test cubes is encoded into a seed and a polynomial identifier. In order to generate a group of deterministic patterns, the LFSR is loaded with a seed and the feedback interconnections are programmed with the polynomial identifier. Then, the LFSR is cycled to generate each of the cubes in the group. This technique can also be used to implement a mixed-

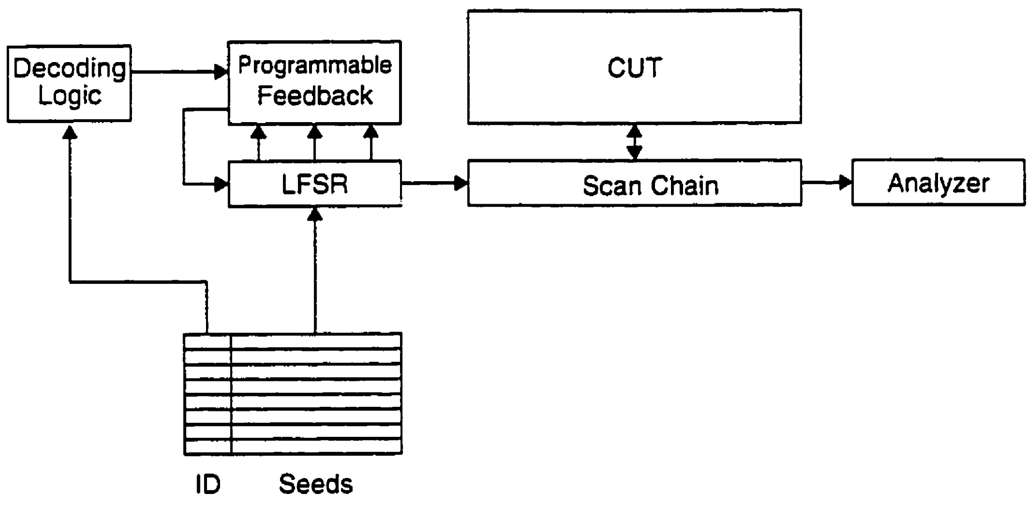


Figure 3.5: Decompression scheme based on Multiple Polynomial LFSRs

mode generator [23, 48]. Using this technique, the MP-LFSR generates pseudo-random patterns to cover the easy-to-test faults in the circuit. The same MP-LFSR then generates the deterministic patterns.

MP-LFSR reseeding has the potential to greatly reduce the amount of test data required to test scan-based designs, but may require significant area overhead. It was shown that a test cube with s specified bits is encoded with approximately s bits. For the distributions shown in Fig. 3.4, this would corresponds to a reduction of a factor greater than six since test cubes have fewer than 15% of their positions specified. However, the technique assumes that a MP-LFSR is implemented on the chip. In order to implement the MP-LFSR, many flip-flops and XOR gates may be necessary.

3.6 Scan and Built-In Self Test

Several strategies have been proposed to implement built-in self test for scan-based circuits. These strategies are qualified as *test-per-scan* or *test-per-clock* [4]. For test-per-scan techniques, the test pattern generator re-loads the scan chains serially for each pattern. Hence, only one pattern for each scan reload is applied. In contrast. test-per-clock techniques allow for one pattern to be applied at each clock cycle. The test pattern generator does not reload the scan chains for each pattern. The focus here is on test-per-scan approaches.

BIST schemes incorporate a test pattern generator and a test response analyzer. Let us consider the implementation of a test pattern generator for BIST.

3.6.1 Test Pattern Generators for BIST

One objective of any BIST scheme is to incorporate a test pattern generator according to the following requirements. A test pattern generator should generate patterns that cover all faults and require: (1) low area overhead; (2) acceptable test application time; and (3) acceptable memory requirements.

Several generators have been proposed for BIST.

ROM-based generators

The simplest generator of test patterns is composed of a ROM unit and a counter [3]. The ROM stores a set of deterministic patterns that covers all faults in the circuit, i. e., the ROM stores a complete test set for the circuit. The counter is used during test application to step through all the valid ROM addresses. For each address supplied by the counter, the ROM drives on its output a new pattern that is applied to the circuit under test.

Even though this type of generator achieves complete fault coverage in an acceptable number of patterns, it is not feasible for most scan-based circuits because of the huge memory requirements associated with deterministic patterns. Even for small circuits, the size of the ROM may be prohibitive. As an example, the circuit used as example in Chapter 1 would require a 52Mbyte- ROM to store the test patterns.

Exhaustive Generators

Exhaustive pattern generators generate all possible patterns that can be applied to a circuit. For a circuit with n inputs, all 2^n possible patterns are applied. A simple binary counter or Gray-code generator are examples of exhaustive generators.

This type of generator guarantees detection of all testable faults. Furthermore, it can be implemented with moderate area overhead. However, the number of patterns that have to be generated is prohibitive.

Pseudo-Random Generators

Linear sequential circuits are often used to generate binary sequences. By imposing some constraints, the generated sequences, although deterministic and repeatable, pass some important randomness properties [18]. Hence, these circuits are often used to generate pseudo-random patterns.

The most popular pseudo-random generator is the *linear feedback shift register* [18]. The general theory of LFSR will be covered in Chapter 4. Other pseudo-random generators comprise *cellular automata* circuits [24], and more complex structures, like *GLFRS*s [36] and *segmented LFSR*s [26].

Pseudo-random generators, in general, do not require excessive area overhead, and integrate well with scan. However, since the patterns are random, it is often difficult to achieve complete fault coverage. In fact, most circuits have random-pattern resistant faults, i. e., faults that have low probability of being covered by random patterns. For these circuits, it is impossible to achieve complete fault coverage with a pseudo-random generator unless some circuit modifications are imposed [44].

Hybrid Approaches

Hybrid or mixed-mode generators combine two types of patterns to achieve complete fault coverage in an acceptable number of patterns while requiring acceptable memory requirements. The most popular type of mixed-mode generators use pseudo-random and deterministic patterns. The pseudo-random patterns cover most of the faults in the circuit. The deterministic patterns cover the remaining undetected faults.

MP-LFSRs are example of a mixed-mode generator. When clocked autonomously, they generate pseudo-random sequences that are used to cover most faults with very little memory (only the initial seed). Then, the remaining faults are covered with deterministic patterns. The deterministic patterns are generated by loading the MP-LFSR with pre-calculated seeds and clocking it autonomously to apply one pattern. Weighted random pattern generators are also examples of mixed-mode generators. Sequences with weight 0.5 are used to cover the easy-to-test faults. Then, the generator uses weight information to bias the generated sequences to generate deterministic patterns to cover the hard-to-test faults.

Yet, mixed-mode generators tend to require excessive area overhead. Generators for weighted random patterns required four flip-flops for each scan input and bias circuitry. The overhead in this case tends to be unreasonable. Similarly, generators based on MP-LFSRs tend to require a large number of flip-flops and XOR gates.

3.7 Motivation of Thesis

High test quality levels dictate the use of pattern generators that achieve complete fault coverage with moderate costs. These requirements are valid for circuits with BIST, for which the test pattern generator is on-chip and those without BIST, for which an external tester generates the patterns.

The requirements are not met by pseudo-random pattern generator (used in autonomous mode only). As was seen, linear feedback shift registers, cellular automata, and other linear circuits, cannot achieve complete fault coverage in an acceptable number of patterns without imposing numerous circuit modifications because of random-pattern-resistant faults.

On the other hand, mixed-mode generators are very promising since they achieve complete fault coverage with moderate memory volumes and test application times. Two types of mixed-mode generators were proposed: weighted pattern generator and MP-LFSRs. Weighted pattern testing greatly reduces memory volumes but the memory requirements still tend to be enormous. Furthermore, the area overhead to implement the scheme makes it impossible to use for BIST applications where the generator is implemented on chip. Reseeding of MP-LFSR is an attractive method to compress deterministic test cubes. However, the MP-LFSR required to generate patterns tends to be large. Consequently, implementations of the MP-LFSR may impose significant area overhead. Furthermore, the technique was investigates only for circuits with a single scan chains. No results or feasibility study have been reported for circuits with multiple scan chains.

Motivation of Thesis

This thesis proposes a new technique to generate deterministic patterns while requiring little area and little memory requirements. The technique is based on the reseeding of an LFSR but uses variable-length seeds to compress deterministic patterns. Each deterministic pattern is encoded into a seed whose length is allowed to be smaller than the length of the LFSR and is allowed to vary from pattern to pattern. It will be shown that variable-length reseeding, even though it encodes each deterministic pattern separately, achieves encoding results equal or better than schemes based on concatenation [52].

The decompression hardware can be implemented with minimal area overhead by sharing flip-flops with the scan or other modules. The circuit under test is already composed of many flip-flops. By re-using these flip-flops, it is possible to implement the decompressor without adding any new flip-flops. The sharing of flip-flops with the scan chain is not possible for techniques that rely on concatenation. This is because the content of the MP-LFSR has to be preserved during the applications of each test pattern in a test group. If scan flip-flops were used, the content of the MP-LFSR would be overwritten while the response of the circuit is loaded into the scan chain.

The proposed scheme can be used for external testing in which the seeds are stored in the memory of the test equipment. The device under test have on-chip decompressor which decompresses the deterministic patterns as they are transmitted to the chip. Hence, the scheme reduces the amount of test data to store in the tester. The amount of data transmitted to the device under test is also reduced since decompression occurs on the chip. The proposed scheme can also be used to design and implement a mixed-mode generator for built-in self test. Test pattern generation is one of the objective of any BIST scheme. For BIST, memory is often the most expensive resource and the capacity between modules may be limited. By using the technique introduced in the thesis, it is possible to design a mixed-mode generator that achieves complete fault coverage with acceptable memory requirements.

The scheme targets scan-based designs with Signature Analysis as shown in Fig. 3.6.

Motivation of Thesis

It can be applied to circuits with many scan chains. It is not restricted to designs with a single scan chain.

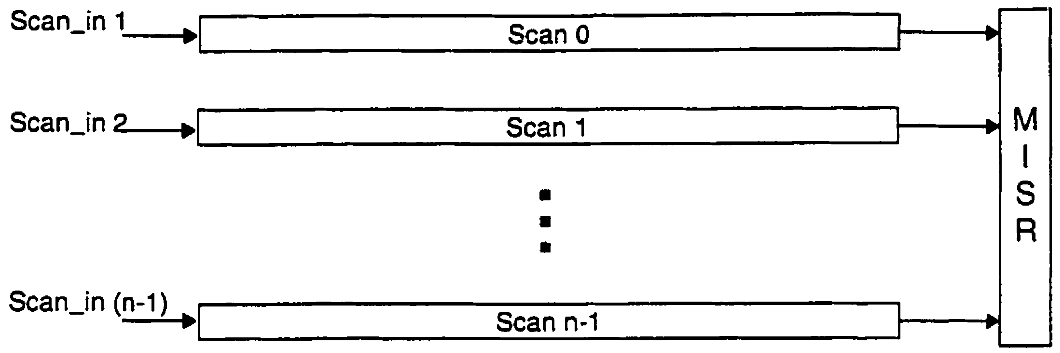


Figure 3.6: Multiple scan design with Signature Analysis

Chapter 4 *Encoding Test Cubes as Variable-Length Seeds of a Linear Feedback Shift Register*

4.1 Introduction

Linear digital circuits form an important class of circuits. They are extensively used to compress and decompress information, to generate codes, especially error-correcting codes, to encode information in cryptography applications, and to implement self-checking circuitry [18]. Linear circuits are composed of only two components: (1) D flip-flops, and (2) Modulo-2 adders, i. e., XOR gates. These components can all be implemented in hardware with modest area requirements.

A linear circuit, whether combinational or sequential has interesting properties. If the circuit is combinational and implements a function $f(a_0, a_1, ..., a_{k-1})$, the function f can always be expressed as $h_0a_0+h_1a_1+...+h_{k-1}a_{k-1}$ (mod 2), where the h_i's are binary constants. If the circuit is sequential with binary vector $V(0) = (a_0, a_1, ..., a_{k-1})$ representing the initial state of the circuit, then, the state of the circuit after the i^{th} cycle can be obtained from the following formula:

$$\begin{bmatrix} X_0(i) \\ X_1(i) \\ \dots \\ X_{k-1}(i) \end{bmatrix} = M^i \begin{bmatrix} X_0(0) \\ X_1(0) \\ \dots \\ X_{k-1}(0) \end{bmatrix} \pmod 2 \qquad (1)$$

where $M_{k,k}$ is the companion matrix of the circuit.

An important sub-class of linear circuits are *linear feedback shift registers*. This structure is very often used to generate test patterns for scan-based circuits because its implementation requires moderate area costs, its shifting properties integrate well with scan chains and the sequence of 0s and 1s it produces pass several randomness tests.

This chapter demonstrates that LFSRs can be used as a mechanism to compress and decompress deterministic test cubes. The main focus is to introduce the concept of variable-length reseeding which is a novel technique used in Chapter 5 to generate deterministic patterns to test scan-based designs.

4.2 Basic Theory of LFSRs

A *shift register* is a series of flip-flops with one input and one output. The flip-flops are arranged such that, at each cycle, the content of each flip-flop is shifted to the next one, while the value of the input is loaded into the first flip-flop. The output is connected to the last flip-flop and constantly monitors its value. Fig. 4.1(a) shows an example of a 5-bit shift register.

A *linear feedback shift register*, or LFSR, is a shift register with a special linear function of the state flip-flops connected to its primary input. This special linear circuit is called the *feedback network* of the LFSR. If the initial states of the flip-flops are denoted by a_0, a_1, ..., a_{k-1}, the feedback function of the LFSR can always be expressed as $h_0a_0 + h_1a_1 + ... + h_{k-1}a_{k-1}$ (mod 2), where the h_i's are binary constants.

There are three important definitions related to LFSRs that will be used extensively in this thesis. The *length* of the LFSR is the number of flip-flops it is composed of. A *feedback* or a *feedback tap* is a connection from the output of an LFSR flip-flop to the feedback network. Finally, the *number of feedbacks* is the number of such connections.

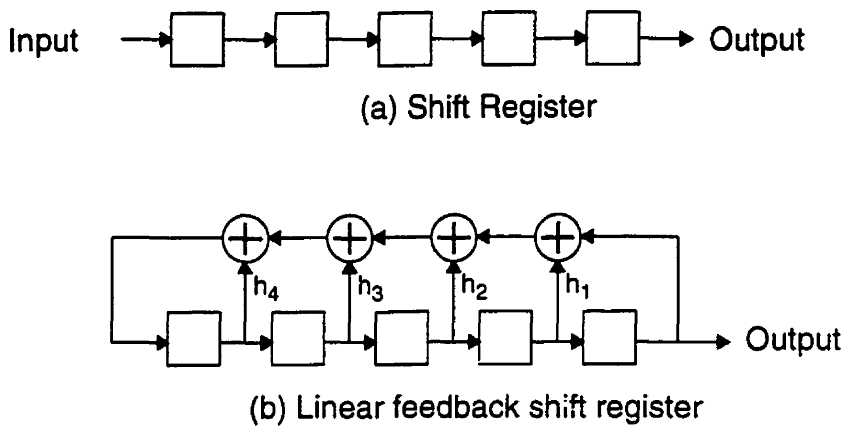


Figure 4.1: Shift Register and LFSRs

The output of the LFSR at successive time steps forms a periodic sequence of 0s and 1s. The *period* P of a k-stage LFSR is always smaller than or equal to 2^k-1. Sequences with period P=2^k-1 are called *maximum-length sequences*, or *m-sequences*. These sequences pass several randomness tests; thus, they are often called pseudo-random sequences. The following lists some of these randomness properties:

Property I. The number of 0s and 1s is almost the same. In fact, the number of 1s differs from the number of 0s by one.

Property II. One half of the runs have length 1, one fourth have length 2, one eighth have length 3, and so on.

Property III. (Autocorrelation property) Any shifted version of the sequence agrees in 2^{n-1} - 1 positions and differs in 2^{n-1}.

4.2.1 Linear Recurrence

The sequence produced by a k-bit LFSR is denoted symbolically by a_0, a_1, a_2, and so on, where the first k-bits $(a_0, a_1, ..., a_{k-1})$ are the initial values of the LFSR flip-flops. The set of initial values $A = (a_0, a_1, ..., a_{k-1})$ is called the ***seed*** of the LFSR.

The feedback function defines a recurrence relation between the output bits. For an LFSR with k stages, the recurrence relation can be expressed as

$$a_{i+k} = a_0 + h_1 a_{i+1} + h_2 a_{i+2} + \dots + h_{k-1} a_{i+k-1} \pmod 2 \tag{2}$$

where the h_i's are binary constants.

4.2.2 Characteristic Polynomial

Let X be the delay operator. Then, the recurrence relation defined by the LFSR can be expressed as

$$X^k a_i = a_i + h_1 X^1 a_i + h_2 X^2 a_i + \dots + h_{k-1} X^{k-1} a_i \pmod 2 \tag{3}$$

$$\text{or,} \left(1 + \sum_{i=1}^{k-1} h_i X^i + X^k\right) a_i = 0 \pmod 2 \tag{4}$$

where $1 + \sum_{i=1}^{k-1} h_i X^i + X^k$ is the *characteristic polynomial* of the LFSR. It can be seen that the degree of the characteristic polynomial is equal to the length of the LFSR. Furthermore, the number of terms in the polynomial is equal to the number of feedbacks in the LFSR minus 1.

In fact, the characteristic polynomial completely defines the structure and the properties of the LFSR. By imposing constraints on the characteristic polynomial of an LFSR, it is possible to guarantee certain properties about the generated sequence. *Primitive polynomials* over GF(2) are of particular interests here [18].

Theorem 4.1: A polynomial h(X) is primitive if and only if the sequence produced by an LFSR with characteristic polynomial h(X) has maximum length. [18]

Theorem 4.2: A polynomial of degree k is primitive if and only if it is irreducible (mod 2) and it divides the polynomial $X^{2^k - 1} + 1$ *(mod 2). [18]*

Theorem 4.3: There is at least one primitive polynomial for every degree. In fact, the number of primitive polynomial of degree k is equal to $E(2^k\text{-}1)/k$ *where E() is the Euler function. Table 4.1 gives the number of primitive polynomials for degrees 1, 2, 4, 8, 16 and 24. [18]*

Degree	Number of primitive polynomials
1	1
2	1
4	2
8	16
16	2048
24	276 480

Table 4.1: Number of primitive polynomials (over GF(2))

4.2.3 Relation between Seed and Generated Sequence

Fig. 4.2 shows a 3-bit LFSR with characteristic polynomial X^3+X^2+1 and with initial seed A= (a_0, a_1, a_2). The m-sequence produced by the LFSR has a period equal to 7, and hence, only the first seven bits generated by the LFSR are kept. The figure lists all the combinations of initial seeds and enumerates the corresponding sequences.

As can be seen, there is a one-to-one relationship between the initial seed of the LFSR and the generated sequence. Given the seed, the sequence is completely specified, and vice versa.

Since an LFSR is a linear circuit, the relationship between the seed of the LFSR and the generated sequence can be expressed mathematically as follows

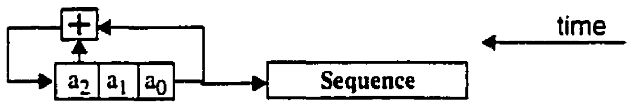
$$a_i = M^i \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} \bullet \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \pmod 2 \qquad (5)$$

where • is the dot product, M is the companion matrix of the LFSR. Equations (5) follows directly from equation (1). The dot product is used to select the sequence of the first flip-flops which generated the LFSR sequence.

The companion matrix of an LFSR with characteristic polynomial $1 + \sum_{i=1}^{k-1} h_i x^i + x^k$ is

$$\begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & 0 \\ 1 & h_1 & h_2 & \dots & h_{k-1} \end{bmatrix} \qquad (6)$$

Equation 5 is linear, and hence, the relationship between the seed and the generated sequence is one-to-one and linear.



LFSR Seed		Sequence
000		0000000
001		0111001
010		1110010
011	↔	1001011
100		1011100
101		1100101
110		0101110
111		0010111

Figure 4.2: Linear, one-to-one relationship between seed and LFSR sequence

4.3 LFSR-Coded Cubes

The one-to-one linear relation between the seed of an LFSR and its generated sequence can be used to compress and decompress information. For example, each 7-bit sequence in Fig. 4.2 can be compressed into a 3-bit seed. Later, these sequences are decompressed or recovered with two main LFSR operations. First, the LFSR is initialized with the appropriate seed. Then, it is clocked for seven cycles.

The number of LFSR sequences is much smaller than the total number of possible sequences which limits the applications of this method to certain particular situations. In Fig. 4.2, the number of LFSR sequences is equal to 2^3, or 8, whereas the total number of possible 7-bit sequences is 2^7, or 128. Clearly, it can be seen that a large fraction of 7-bit sequences cannot be encoded this way.

However, the compression capabilities of LFSRs can be used to compress the deterministic test cubes generated by automatic test pattern generation algorithms. The resulting cubes often have large proportions of unspecified positions, positions that can be either 0 or 1, which makes it possible to encode them as seeds of an LFSR.

Before the mechanism used to encode cubes is described, let us review the definitions and concepts related to cubes. Formally, a *cube* is a sequence of 0s, 1s, and don't cares which are denoted by the symbol x. The sequences (1, 1, x, x), (1, 1, 0), (x, 0, x, 1) and (0, x, 0, 0) are all examples of cubes. The positions, or bits, in a cube may be *specified* or *unspecified*. A specified position is either a 0 or a 1, whereas an unspecified position is a don't care. Hence, the cube (0, x, x, 1, x) has two specified and three unspecified positions. Furthermore, a cube C = $(c_0, c_1, ..., c_{L-1})$ *covers* a binary sequence B = $(b_0, b_1, ..., b_{L-1})$ if and only if, for all position i in the sequence, $c_i = b_i$ or c_i=x. For example, the cube (1,x,x) covers the binary sequences (1,0,0), (1,0,1), (1,1,0), and (1,1,1) but does not cover the sequence (0, 1, 0).

A cube C is compressed into an LFSR-seed A = $(a_0, a_1, ..., a_{k-1})$ by imposing that the sequence generated by the LFSR is covered by the cube C. This is equivalent to imposing that, for each specified position c_i in C, an equality of the form of equation 5, i. e.,

$$c_i = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & 0 \\ 1 & h_1 & h_2 & \dots & h_{k-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{k-1} \end{bmatrix} \bullet \begin{bmatrix} 1 \\ 0 \\ 0 \\ \dots \\ 0 \end{bmatrix} \pmod 2 \qquad (7)$$

This set of equations is linear. Therefore, to obtain the seed, a system of linear equations has to be solved. If the system is *consistent*, at least one seed can be obtained. However, if the system is *inconsistent*, no solution exists and the cube cannot be compressed. Section 4.3.3 will discuss how to guarantee (with arbitrary large probability) that a seed can be obtained.

4.3.1 Example

Let us consider the example of Fig. 4.3 in which a 3-bit LFSR generates 7-bit sequences. The characteristic polynomial of the LFSR is equal to X^3+X^2+1 and is primitive. In the example, a sequence covered by the cube C = (x,x,1,x,0,1,x) is to be generated by the LFSR. The first column shows the LFSR sequence as a function of the seed A = (a_0, a_1, a_2). The second column lists the positions of the cube C. Combining the first two columns creates a system of linear equations (modulo 2). Solving the system yields a seed, which is shown in the third column. If the LFSR is loaded with the seed A = (0,1,1), then the generated sequence is (0,1,1,1,0,1,0) which is covered by C. Hence, the 7-bit cube C can be encoded as a three-bit seed.

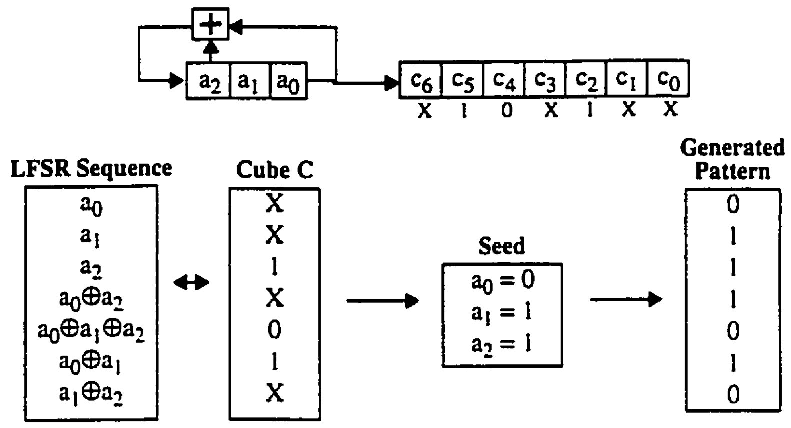


Figure 4.3: Example of LFSR-coded cube

4.3.2 Solving System of Linear Equations (modulo 2)

Gaussian Elimination with maximum pivot is an effective algorithm to solve systems of linear equations modulo 2 [6, 34]. The algorithm begins by forming an augmented matrix M corresponding to the system of equations. Then, it puts the matrix in row echelon form. Finally, it uses backward substitution to obtain one solution.

A binary matrix M is in row echelon form if and only if it satisfies the following criteria: (1) each non-trivial row has a leading 1. A leading entry is the first non-zero entry in a row; and (2) all elements below a leading 1 but in the same column are 0s.

There are well known algorithms to put a matrix in row echelon form [6]. These algorithms use two types of elementary operations to transform the matrix without affecting the final solution. The first operation consists of swapping two rows of the matrix. The second operation consists of adding one row to another. Because all coefficients are binary (mod 2), adding rows can be done using bit-wise XOR instructions.

Fig. 4.4 summarizes the algorithm used to reduce a matrix in row-echelon form. In total, the algorithm required O(n^2) bit-wise XOR operations, where n is the number of equations.

Once M is in row-echelon form, the solution can be read directly. For each row, the leading variable is assigned the value on the right-hand side.

Fig. 4.4 shows the complete procedure to calculate the seed for the example of Fig. 4.3. Combining the m-sequence of the LFSR with cube C creates a system of linear equations. This system is then represented mathematically with an augmented matrix M. The matrix is then put in reduced echelon form, from which two different solutions are obtained. Finally, one solution is chosen to encode cube C.

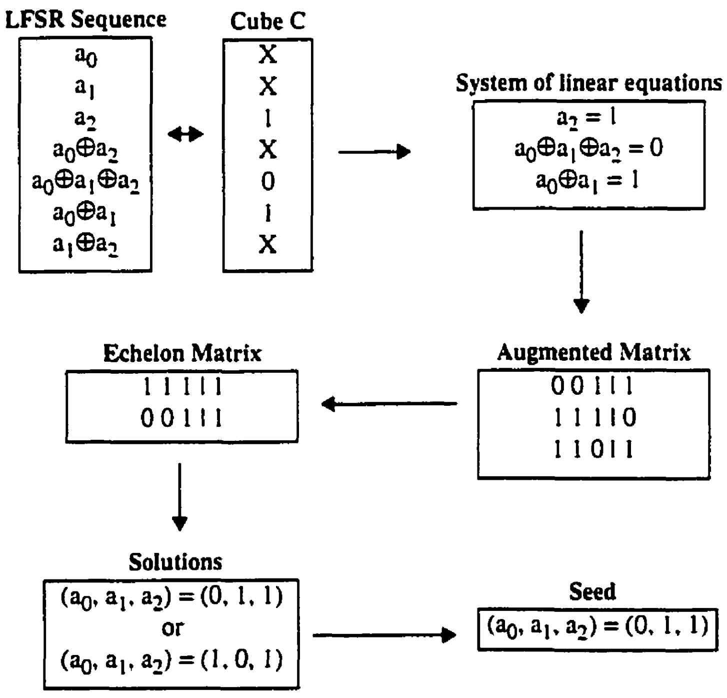


Figure 4.4: Solving systems of linear equations (modulo 2)

4.3.3 Probability of Existence of a Solution

Systems of linear equations (mod 2) have either one, many or no solutions.

While encoding a cube into an LFSR seed, the probability that the system of equations has at least one solution depends on many factors [22]. The number of flip-flops in the LFSR determines the number of variables. The number of specified bits in C determines the number of equations. The length of the cube, the position of the specified bits in it, and the characteristic polynomial have an impact on the dependence between the equations in the system.

If the characteristic polynomial $h(X)$ is primitive and has five or more terms, and the period of the LFSR is much larger than the length of the cube, the probability of solving the resulting system of equations, P_{sol}, is mainly a function of the difference between k, the degree of $h(X)$ and s the number of specified bits in C [21-23].

The probability P_{sol}, was estimated for a number of cases [21]. The results are summarized in Table 4.2.

$k - s$	P_{sol}
-4	0.06
-2	0.22
0	0.61
1	0.78
2	0.88
4	0.97
8	0.998
16	> 0.9999
20	> 0.999999

Table 4.2: Theoretical values of P_{sol} as a function of k, the length of the LFSR, and s, the number of specified positions in the cube.

By using the results of Table 4.2, the length of the LFSR is chosen to make the probability of encoding as high as required. In fact, by selecting the length of the LFSR to be 20 units larger than the number of specified positions in the cube, the probability of encoding is greater than 0.999999. Also, from Table 4.2, probabilities of encoding for specific cases can be obtained. For instance, there is a 61% probability to encode a cube containing 32 specified bits with a LFSR of length 32.

4.4 Encoding Sets of Cubes

An LFSR can be used as a mean to compress a set of test cubes into seeds. Later, the test cubes are re-generated by loading the content of the LFSR with the pre-calculated seeds. This approach of generating patterns is called ***reseeding*** [28] since the seed of the LFSR is changed to generate deterministic patterns. However, reseeding should not be confused with the approach of [41] in which the seed of the LFSR is changed randomly to improve the randomness of the LFSR sequence.

In order to measure the efficiency of the encoding, a measure called *encoding efficiency* is used [22]. This measure is defined as the ratio of the average number of specified bits in the test cubes over the average number of bits to store the seeds. The encoding efficiency is expected to be between 0 and 1. The higher the efficiency, the fewer bits have to be stored.

There are three main mechanisms to encode a set of deterministic test cubes: (1) encoding each test cube separately; (2) encoding groups of test cubes; and (3) encoding each test cube as a variable-length seeds. The latter is the novel technique proposed in this thesis.

Given a set of test cubes, one can encode them separately [28]. The length of the LFSR has to be selected such that all test cubes can be encoded. Since the probability is inversely proportional to the number of specified bits in a test cube, the length of the LFSR is usually adjusted such that the test cubes with the most number of specified bits can be encoded into a seed. Hence, the length of the LFSR is determined directly by the maximum number of specified bits found in a cube.

With this approach, the deterministic test cubes are compressed as seeds of a pre-determined LFSR. The encoding efficiency is simply equal to s_{avg} / k, where s_{avg} is the average number of specified bits in a test cube and k is the length of the LFSR.

This approach has one important drawback that arises when the cubes in the set have high variation in the number of specified bits. For these cases, cubes with few specified bits will not be encoded efficiently [52] and tend to lower the average number of specified positions in the test cubes. As a consequence, s_{avg} is often much lower than the length of the LFSR required. Therefore, the encoding efficiency of this scheme tends to be low.

Another approach to compress a set of cubes is to exploit the fact that the length of the generated sequence is independent of the length of the cubes. The idea is to *concatenate* the cubes in groups [21-23, 48]. Each group rather than each cube is encoded.

The cubes are concatenated such that the total number of specified bits in each group is approximately equal. Hence, the compression obtained tend not to depend on the variability of the number of specified bits. However, the length of the decompression LFSR used to generate the cubes has to be adjusted such that the test group with the highest total number of specified bits can be encoded. Therefore, the length of the LFSR required by concatenation techniques tends to be larger.

The encoding efficiency of methods based on concatenation is equal to s_{gavg} / k, where s_{gavg} is the average number of specified bits in a test group, and k is the length of the LFSR. By using clever test set processing techniques [22], it is possible to make s_{gavg} close to k. Hence, the encoding efficiency of these methods is very close to 1.

However, concatenation imposes an important requirement: the content of the LFSR cannot be overwritten while it generates the cubes within one group. This additional requirement prohibits sharing of hardware structure already on the chip. In particular, it is impossible to implement the LFSR by re-using flip-flops already on the chip (see Chapter 5). In general, schemes based on concatenation tend to require many flip-flops, which may impose significant area overhead.

Another method to encode a set of cubes is to use *variable-length seeds*. The next section is devoted to this technique.

4.5 Variable-Length Seeds

One very efficient method to generate a set of deterministic test cubes would be to adjust the length of the LFSR such that each cube is encoded with as few bits as possible. However, this approach would be difficult to implement since the number of specified bits varies greatly from pattern to pattern.

Yet, the same effect is achieved by using seeds of varying lengths. The *length* of a seed is equal to n if and only if it can be expressed as $(0, 0, ..., 0, 1, a_{k-n+1}, a_{k-n+2}, ..., a_{k-1})$. Similarly, a seed has a length shorter or equal to n if it can be expressed as $(0, 0, ..., 0, a_{k-n}, a_{k-n+1}, ..., a_{k-1})$. For example, the seed A = (0, 0, 1, 0, 1) has length 3, while seed A = (0, 0, 0, 1) has length 1.

From the above definition, a seed is divided into two parts. The first part consists of the first $(k - n)$ bits which are 0 by definition. The second part is non-trivial and is composed of the seed variables a_i's.

If the content of the LFSR is assumed to be initially reset, only the second part of the seed has to be memorized. The other part of the seed does not have to be stored since it is known to be 0. For instance, a seed of length n (or shorter) is stored as $(a_{k-n}, a_{k-n+1}, ..., a_{k-1})$ independently of the length of the LFSR.

The overall method is called ***variable-length reseeding*** since the lengths of the seeds are adjusted to encode the test cubes.

4.5.1 Example

Let us re-consider the example of Fig. 4.3. The seed was calculated to be (0, 1, 1) which is a seed of length 2. Assuming that the LFSR is reset prior to loading the seed, only the last two bits (1, 1) have to be remembered. Thus, the 7-bit sequence (x, x, 1, x, 0, 1, x) can be compressed into a 2-bit seed.

4.5.2 Expected Length of a Seed

One very important parameter of variable-length reseeding is u, the expected length of the seeds. On the average, a cube with s specified bits can be encoded with a seed of length u. Hence, with the knowledge of the expected length of the seeds, it is possible to estimate the encoding efficiency as the ratio s / u.

The following discussion develops a probabilistic model, based on [21] and [22], that will establish $P(n, s)$, the probability that a test cube with s specified bits can be compressed with a seed of length *n or shorter*. From this probabilistic model, the expected value of the length of the seed is obtained and is used to calculate the encoding efficiency of variable-length reseeding.

Fig. 4.5 illustrates the procedure to calculate the seed associated with the test vector (x, x, 0, 1, x, x, 0) obtained from a 3-bit LFSR with the polynomial $X^3 + X^2 + 1$ *if* the last bit of the seed is assumed 0. The sequence of the LFSR in terms of the initial seed variables is equated to the test pattern, which yields a system of 3 equations because there are 3 specified bits in the test vector. The system of equations is consistent. Consequently, a seed of the form $(0, a_1, a_2)$ exists for the test vector.

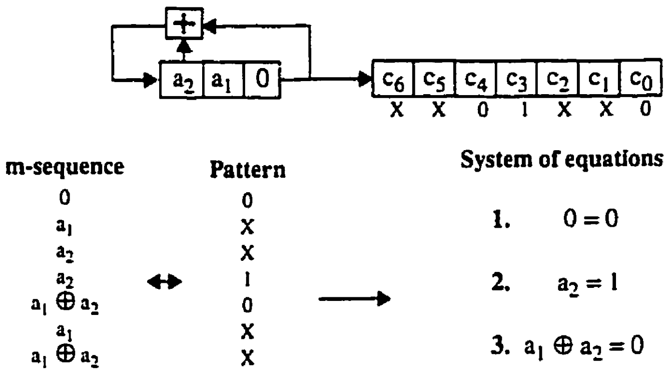


Figure 4.5: Example of procedure to calculate seeds with some positions assumed 0.

From the example of Fig. 4.5, two observations can be formulated about the m-sequence of the LFSR if some seed variables assume the value of 0. First, there are some positions in the m-sequence that are always 0 no matter what the value of the seed variables a_1 and a_2 are. Secondly, the m-sequence contains all the linear combinations of the variables a_1 and a_2 and every distinct linear combination appears more than once.

These two observations can be generalized to the case of a k-bit LFSR (associated with a primitive polynomial) loaded with a seed of the form $(0, 0, ..., 0, a_{k-n}, a_{k-n+1}, ..., a_{k-1})$, where n is the number of variables not assumed 0. For this general case, the number of constant 0s in the m-sequence is $2^{k-n} - 1$ and each distinct linear combination appears 2^{k-n} times in the m-sequence.

In order to derive $P(n, s)$, let us consider the process of forming the equations to calculate the seed for a test vector with s specified bits. At every step, a new equation is formed and the probability that the system of equations remains consistent is calculated. The process starts with an empty set of equations and stops after the s equations are formed.

This process can be described by a graph G in which the vertices represent consistent systems of equations and the edges represent transitions that preserve the consistency of the system as a new equation is formed. In G, the vertex $X_{t,d}$ denotes a consistent system of t equations with rank d. Every edge is labeled with a weight representing the probability of the corresponding transition. The weights are functions of the rank d of the system.

Fig. 4.6 shows the graph G. The process of forming the equations starts at the vertex $X_{0,0}$. As new equations are created, the state of the system changes and if it remains consistent, it is represented by one vertex of the graph. Two transitions that maintain the consistency of the system are possible since the next equation added may or may not increase the rank of the system. $\alpha(d)$ is the probability that the next equation does not increase the rank but preserves the consistency. $\beta(d)$ is the probability that the next equation increase the rank. For the latter, the consistency of the system is guaranteed to be preserved.

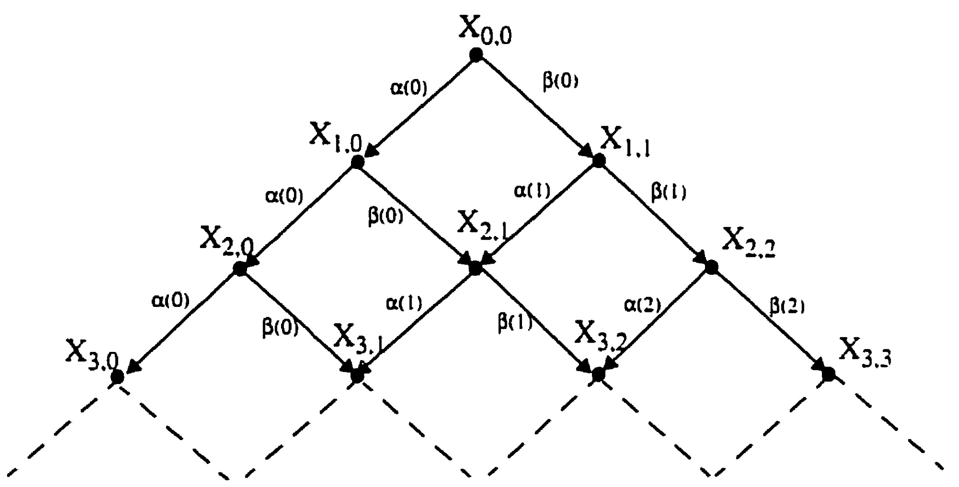


Figure 4.6: Graph G.

To derive the transition probabilities $\alpha(d)$ and $\beta(d)$, let us consider the state $X_{t,d}$ which relates to a consistent system of t equations with rank d. The system of equations comprises t linear combinations of the m-sequence of the LFSR. Since the rank is d, there are 2^d - 1 different linear combinations that are dependent on those contained in the system of equations. These linear combinations all appear 2^{k-n} times in the m-sequence. If the constant 0's are included, the total number of positions in the m-sequence that are linearly dependent on the t linear combinations contained in the system of equations is $(2^d - 1)2^{k-n} + 2^{k-n} - 1$. Since t of these positions are already included in the system of equations, the proportion of dependent positions in the m-sequence but not in the system of equations is

$$\frac{\left(2^d - 1\right)2^{k-n} + 2^{k-n} - 1 - t}{2^k - 1} \approx 2^{d-n} \quad \text{for } 2^k \gg 1 \tag{8}$$

The probability that the next equation formed increases the rank of the system is $1 - 2^{d-n}$. For such case, the consistency of the system is preserved. Hence,

$$\beta(d) = 1 - 2^{d-n} \tag{9}$$

If the next equations added does not increase the rank of the system, the consistency of the system is no longer guaranteed and depends on the constant part of the new equation. Assuming that the specified bits in the test cube are random, the probability that the system remains consistent is 1/2. This implies that

$$\alpha(d) = \left(\frac{1}{2}\right)2^{d-n} = 2^{d-n-1} \tag{10}$$

It is important to note that since some positions in the m-sequence contain constant 0's or similar linear combinations, all states $X_{t,d}$ with $t \geq 0, d \geq 0$ have to be considered even if $t > 2^d - 1$. For instance, the state $X_{2,0}$ represents a consistent system of two equations having rank 0, i.e., the set of equations $\{ 0 = 0, 0 = 0 \}$.

The probability that a seed of the form $(0, 0, ..., 0, a_{k-n}, a_{k-n+1}, ..., a_{k-1})$ exists is equal to the probability that the corresponding system of s equations is consistent. It is equivalent to the probability of reaching the vertices $X_{s,0}, X_{s,1}, X_{s,2}, ..., X_{s,s}$ as they represent all the possible consistent systems of s equations. For each path, the transition probabilities are multiplied together and the final result is the sum of these products. For instance, the probability that a test vector with 2 specified bits can be encoded with a seed of length 2 or shorter is obtained with

$$P(2,2) = \alpha(0)\alpha(0) + \alpha(0)\beta(0) + \beta(0)\alpha(1) + \beta(0)\beta(1) \tag{11}$$

The values for $P(n, s)$ can all be calculated based on this approach. In particular, if $n = s$ and $n > 14$, $P(s, s) \approx 0.61$.

$P(n, s)$ relates to seeds of length n or shorter, not directly to the actual lengths of the seeds. Let N be a discrete random variable that represents the actual length of the seed corresponding to a given test vector with s specified bits. The probability that $N = n$, denoted by $P[N = n]$, can be expressed as

$$P[N = n] = P(n, s) - P(n-1, s) \quad . \tag{12}$$

and the expected value of N, denoted by u, can be calculated with

$$u = E(N) = \sum nP[N = n] \tag{13}$$

For the cases where n and s are larger than 14, $P[N = n]$ is mainly a function of n-s and the expected value for N, obtained from equation (13), is approximately

$$E(N) \approx s \tag{14}$$

From the above derivation, the encoding efficiency is expected to be very close to 1.

The above results can be formalized in a theorem:

Theorem 4.4: Let C be a cube of length L with s specified bits, h(X) be a primitive polynomial of degree k, and assume L is equal to 2^k-1. Then, the expected length of the seed encoding C is equal to s.

Proof: From the above discussion.

Theorem 4.4 is restricted to cubes whose length L is equal to 2^k-1, the period of the m-sequence. This assumption is not very realistic: in most cases, the period of the m-sequence is much larger than the length of the cube. Theorem 4.4 also applies to cases where L is shorter than the period of the LFSR *if* the linear dependencies are somehow evenly distributed in the m-sequence [39].

Linear dependencies in LFSR sequences have been considered in several papers [9, 13]. However, the distribution of linear dependencies in the m-sequence of LFSR has only been considered in [39] where it was shown that linear dependencies in the m-sequence of primitive polynomials with five or more terms tend to be uniformly distributed. Extensive Monte Carlo experiments reported in [52] indicate that irreducible polynomials with five or more terms tend to generate sequences in which linear dependencies are uniformly distributed.

4.6 LFSRs to Generate Parallel Sequences

LFSRs can be used to generate n parallel sequences, which are related to one another by a simple phase shift or delay [8, 26, 36]. By carefully selecting the time shifts between each sequences, it is possible to obtain a set of sequences that are virtually unrelated. To generate parallel sequences, it is necessary to get access to the value of many flip-flops within the LFSR. In the following, it is assumed that the output of each flip-flop can be utilized.

There are two main approaches to generate parallel sequences from an LFSR [8]. The first approach adds a phase shifter function to generate each of the n sequences. The second approach breaks the LFSR into n segments. Each segment generates one of the sequences.

4.6.1 Usage of Phase Shifters

Fig. 4.7 shows the first method to generate n parallel sequences [8]. In essence, the sequences are generated by constructing *n* different linear function of the LFSR state. Each linear function produces a sequence, exactly like the one produced by the LFSR, except for a time shift. The resulting network is called a *phase shifter* and consists of a tree of

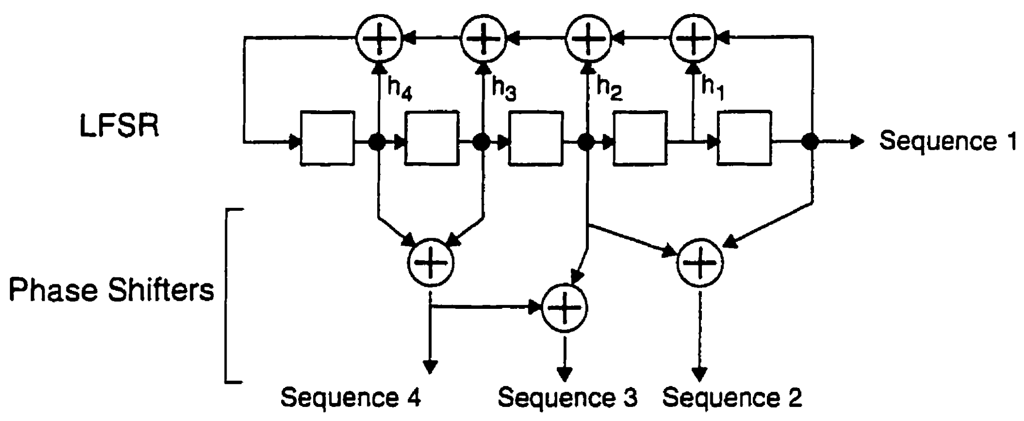


Figure 4.7: LFSR with phase shifters

XOR gates.

4.6.2 Segmentation of LFSR

A *k*-bit LFSR can be broken into *n* interconnected segments, each of length (*k* / *n*) [8, 26]. Each segment is itself an LFSR and is used to generate one sequence. The segments are linked together by adding to their feedback networks taps taken from other LFSRs. For example, Fig. 4.8 shows a 15-bit LFSR broken into 3 segments.

The sequences produced by the *n* segments are shifted version of the sequence produced by an equivalent *k*-bit LFSR [8]. Therefore, usage of phase shifters and segmentation produce equivalent hardware structures.

As an example, let us derive the equivalent characteristic polynomial of the segmented LFSR of Fig. 4.8. In order to derive the polynomial, the recurrence relation created by the

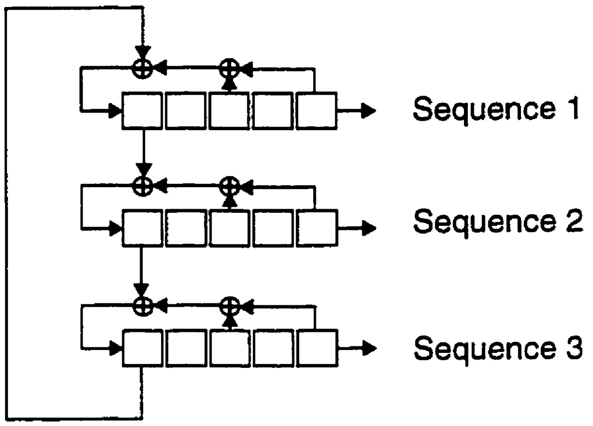


Figure 4.8: Segmented LFSR

feedback network has to be known. Let a_{ij} be the ith bit generated by the jth segment and let X be the delay operator. The recurrence relations can be expressed as

$$\begin{cases} X^5 a_{i0} + X^3 a_{i0} + a_{i0} + X^4 a_{i2} = 0 \\ X^5 a_{i1} + X^3 a_{i1} + a_{i1} + X^4 a_{i0} = 0 \\ X^5 a_{i2} + X^3 a_{i2} + a_{i2} + X^4 a_{i1} = 0 \end{cases} \pmod 2 \tag{15}$$

Alternatively, the relations can be formulated in matrix form, i. e.,

$$\begin{bmatrix} (X^5+X^3+1) & 0 & X^4 \\ X^4 & (X^5+X^3+1) & 0 \\ 0 & X^4 & (X^5+X^3+1) \end{bmatrix} \begin{bmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \pmod 2 \tag{16}$$

The characteristic polynomial of a segmented LFSR is the determinant of the coefficient matrix. In (16), the determinant of the coefficient matrix is $X^{15} + X^{13} + X^{12} + X^{11} + X^{10} + X^9 + X^6 + X^5 + X^3 + 1$. Therefore, the segmented LFSR of Fig. 4.8 is equivalent to a 15-bit LFSR with characteristic polynomial $X^{15} + X^{13} + X^{12} + X^{11} + X^{10} + X^9 + X^6 + X^5 + X^3 + 1$ and a phase shifter.

Chapter 5 *Decompression Scheme and Its Implementation*

5.1 Introduction

As was seen in Chapter 1, deterministic stored-pattern testing is increasingly unattractive for scan-based designs because of its huge memory requirements [11, 28]. Each deterministic pattern has to specify the sequence to be shifted into the scan chains, the values to force at the inputs, the expected response of the circuit and the expected sequence to be shifted out of the scan. For medium to large sized designs, the resulting memory requirements are prohibitive.

However, in most cases, applying deterministic patterns is the only method to obtain complete fault coverage without imposing numerous circuit modifications [11, 28]. Even after millions of random patterns are applied and many test points are added, most circuits will still feature a multitude of undetected faults. For applications where complete fault coverage is a must, a method to generate deterministic patterns without imposing huge memory requirements is of practical importance.

This Chapter proposes a new method to generate deterministic patterns for scan-based designs with Signature Analysis or other compaction scheme. For these circuits, the memory require to store deterministic patterns is dominated largely by the input sequence(s) to be shifted into the scan chains.

In the proposed scheme, deterministic patterns are compressed as variable-length seeds of an LFSR and are stored in a memory unit. Each digital integrated circuit has its own LFSR that is used to decompress and shift the input patterns into the scan chains.

For circuits with built-in self test, the LFSR used to decompress the deterministic patterns can be combined with the test pattern generator already incorporated with the circuit. The resulting module is a mixed-mode generator capable of generating random and deterministic patterns with little memory requirements.

Section 5.2 described the decompression architecture. Sections 5.3 and 5.4 demonstrate how to implement the decompression LFSR in hardware or in software.

5.2 Decompression Scheme

The decompression scheme can be implemented in hardware as part of a built-in test strategy using mixed-mode test vectors. The overall architecture consists of a test controller, a number of ICs with on-chip decompression hardware, and a memory unit (Fig. 5.1). To apply one deterministic patterns, the test controller fetches the compressed test data from memory and transfers it through channels to the ICs. The on-chip test hardware decompresses the data it receives and applies the generated test vector to the circuit under test.

The architecture is very general and can be implemented in many ways. The test controller can be off-chip or can be implemented by an embedded core. The decompression LFSR can be implemented in hardware or emulated by an embedded core. Similarly, the memory unit can be on the chip, on separate ICs, or part of some external test equipment.

5.2.1 Test Data Format

Deterministic patterns are encoded as variable-length seeds of an LFSR.

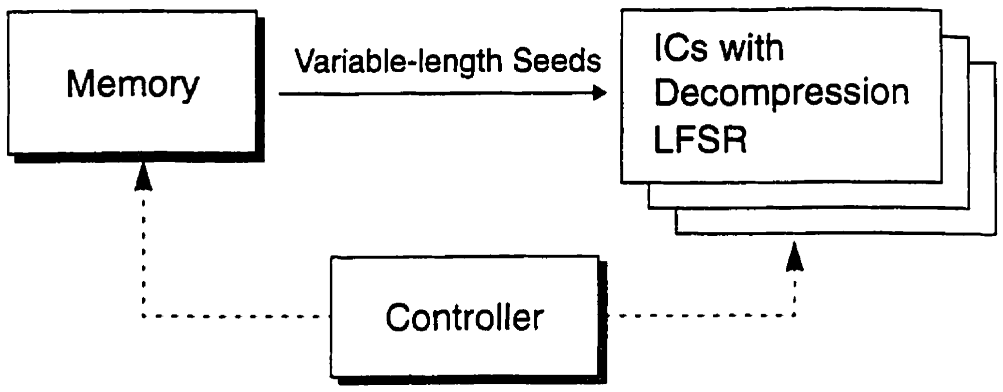


Figure 5.1: Decompression scheme

The encoding efficiency of the scheme depends on the format used to store the test data. Since the seeds have variable-lengths, some additional information must be stored to specify the current length of the seed. In the proposed test data format (Fig. 5.2), it is assumed that the seeds are first sorted in increasing order of length. The format consists of two fields. The *size bit* indicates when to increase the size allocated to store the seed. When the size bit is 1, the size of the seed field allocated for the next encoding is increased by d bits. The *seed field* contains the seed and some extra 0s appended. These extra zeroes are included such that the length of a seed field can always be expressed as $b + i^*d$ where b is the size of the shortest seed.

A set of p patterns can be encoded with $p + n_{total} + v$ bits, where n_{total} is the total number of bits in the seeds, and v is the total number of extra zeroes appended.

The compression index is the ratio of the memory required to store the patterns explicitly over the volume of compressed data. For a circuit with S_{ff} scan flip-flops, the compression index can be expressed as $p^*S_{ff} / (p + n_{total} + v)$. For the cases when n_{total} is significantly greater than p and v, the compression index can be expressed as p^*S_{ff} / n_{total}

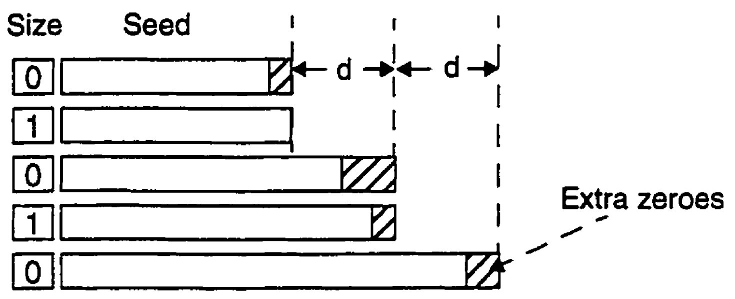


Figure 5.2: Test data format

or $S_{ff} / n_{average}$.

For example, if the test cubes of a circuit with 1000 flip-flops can be encoded into seeds of average length 50, then the compression index would be approximately 20.

5.2.2 Test Controller

The test controller controls the decompression and application of deterministic patterns. This involves reading the test data from the memory unit and transferring it to the on-chip decompression LFSR. In order to read the seeds from memory and load the on-chip LFSR, the test controller has to maintain the current length of the seed. As the controller reads the data, if the size bit is 1, it increases the current length for the next seed by *d* bits. The functionality to maintain the length is implemented with a counter which increments by *d* bits.

5.3 Hardware Implementation

The decompression hardware required to generate deterministic patterns is an LFSR that can be re-initialized serially. However, the LFSR required for deterministic pattern generation tend to require many flip-flops [22, 52, 53]. For some circuits, the number of flip-flops may be almost 20% of the number of flip-flops in the design. Fortunately, the LFSR can be implemented without requiring any additional flip-flops by using flip-flops already on the chip, like scan flip-flops [52, 53]. Furthermore, some circuits have built-in self test capabilities based on the STUMPS architecture. These circuits already contain an LFSR for pseudo-random pattern generation. This LFSR can also be utilized to implement the decompressor [52, 53].

Scan flip-flops can be used to implement the decompression LFSR because the LFSR is re-loaded for each deterministic pattern. Consequently, the content of the LFSR can be overwritten after each pattern is applied to the circuit under test. As was noted in Chapter 3, this property is not featured by schemes based on concatenation, for which the content of the LFSR has to be preserved for a number of patterns.

The next subsections propose various hardware implementation of the decompression hardware. The implementations assume that there is one external pseudo-random pattern generator (PRPG) on the chip, implemented as a type I LFSR. This LFSR would be used in BIST applications to generate pseudo-random patterns. For circuits without BIST capabilities, the LFSR would have to be inserted.

5.3.1 Implementation for circuits with a single scan chain

Fig. 5.3 shows the decompression hardware for a single scan chain. The scheme requires only one extra feedback (controlled by means of an AND gate) from the scan chain, and a multiplexer to allow the seed to be shifted in. During testing, there are two modes of operations: random and deterministic. In the random mode, the extra feedback from the scan chain is disabled and the (BIST) LFSR is used to generate random patterns. In the deterministic mode, the extra feedback from the scan is enabled and two control signals are used to load the seed and perform decompression: signal *Reset* clears the decompression LFSR, while signal *Shift* controls the multiplexer to allow the seeds to be shifted in. The seeds are loaded by first resetting the decompression LFSR and then shifting the seed variables serially through a multiplexer into the LFSR. Once loaded, the seeds are decompressed by exercising the decompression LFSR.

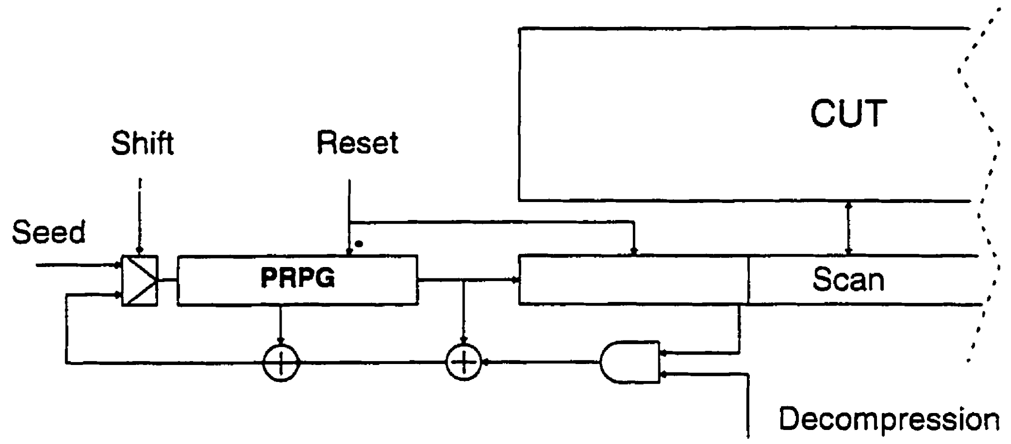


Figure 5.3: Decompression hardware for circuits with a single scan chain.

5.3.2 Decompression hardware for circuits with multiple scan chains

One possible implementation of the decompression hardware for circuit with multiple scan chains is shown in Fig. 5.4. This implementation is the same as the one for circuits with a single scan chain except that a phase shifter (XOR tree) is used to generate more than one bit every clock cycle. The LFSR used for decompression is implemented by using the flip-flops of the PRPG and some scan flip-flops.

However, the scan flip-flops are taken from only one scan chain. The number of flip-flops in one scan chain may not be enough to extend the LFSR. Hence, this implementation may not be adequate for all cases.

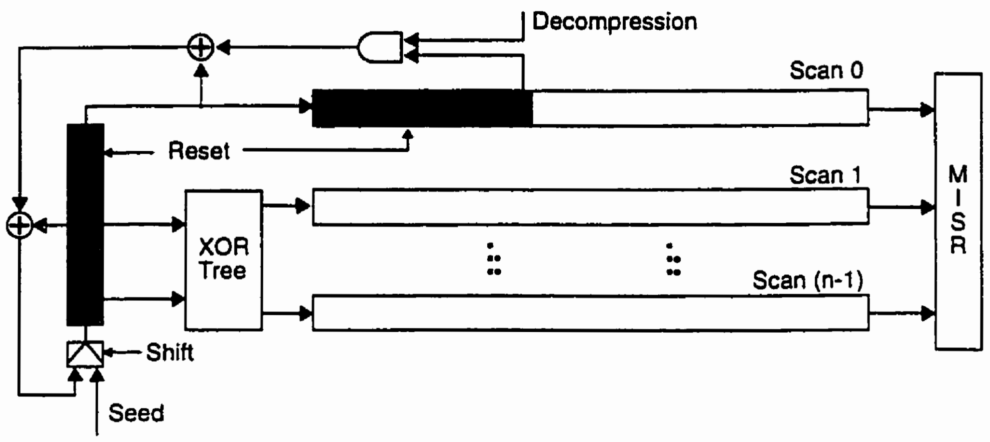


Figure 5.4: Decompressor implemented as an LFSR with phase shifter

Fig. 5.5 shows a more general design of the decompressor for circuits with multiple scan chains.

The decompressor is implemented by adding extra feedbacks from the scan chains and some multiplexers. The feedbacks (shown with dotted lines in the figure) are used to combine the PRPG with scan flip-flops to implement the decompressor. These feedbacks are fed to additional XOR gates between the flip-flops of the PRPG and are controlled by means of an AND gate.

As shown in the figure, a feedback from a scan chain is fed to more than one site in the PRPG. If the scan chains are numbered from 0 to 2^r-1, the feedback from scan number i is connected to positions $(i+2^v) \bmod 2^r$, $v = 0, 1, 2, \ldots, r-1$, of the PRPG.

The same feedback connection from a scan chain is connected to the input of the next scan chain (through a multiplexer). By adding these connections between the scan chains and placing a multiplexer in front of the PRPG, a serial path through the decompressor is created. This serial path is used to load the variable-length seeds.

Notice that the connection between two scan chains can be controlled using an extra

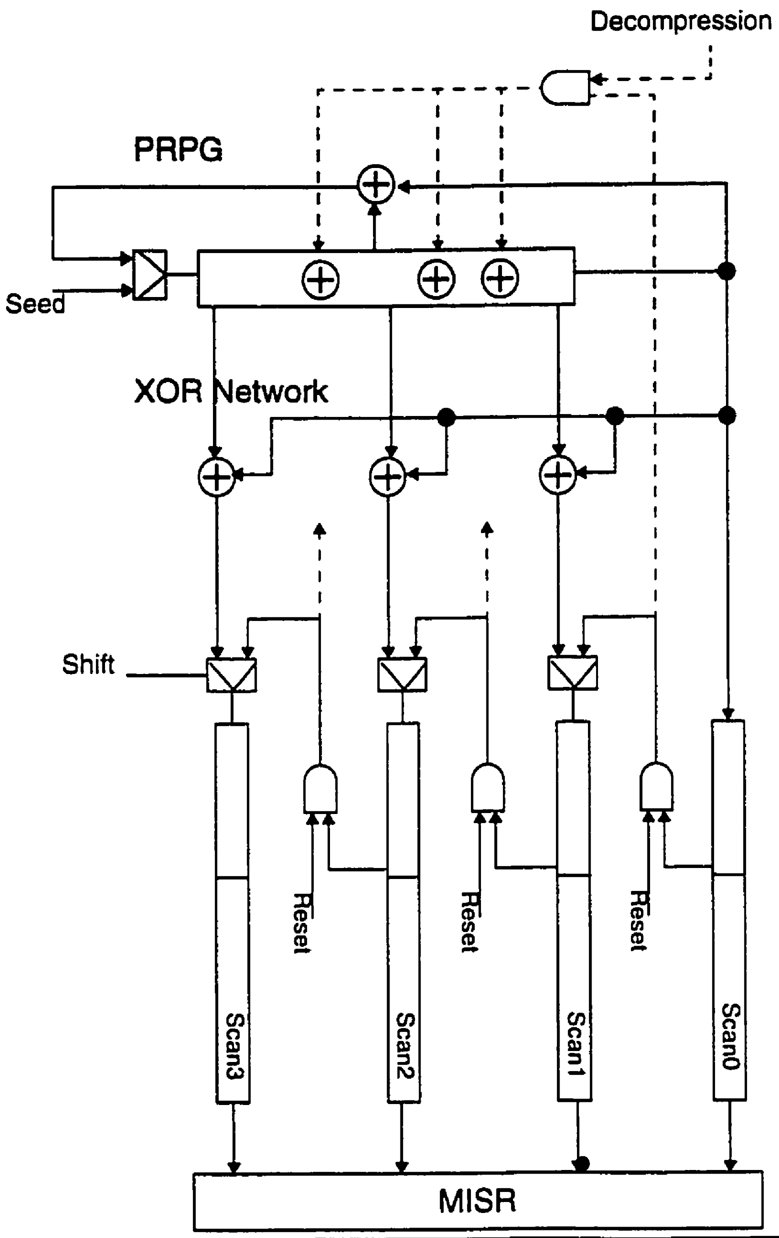


Figure 5.5: Decompression hardware for circuits with multiple scan chains.

AND gate to facilitate reset of part of the seed. Indeed, as the first *s* bits of the seed are shifted, the signal *Reset* can force the value shifted into a scan chain to be 0 and some parts of the decompressor can be reset. Thus, by using this approach, it is possible to load variable-length seeds without having to reset the flip-flops prior to loading each seed.

As in the single scan chain case, the generator has two modes of operation: random and deterministic. In the random mode, the PRPG operates independently and generates pseudo-random patterns. In the deterministic mode, however, the extra feedbacks are enabled to implement the decompressor. By controlling the *shift* signal, the variable-length seeds are loaded one by one and decompressed. The seeds are loaded by shifting the seed variables serially. Once the seed is loaded, decompression is performed in parallel through the XOR network. The signal *Reset* is controlled to reset parts of the seed.

5.4 Software Implementation

Circuits based on data-path architectures constitute an increasingly large portion of integrated chips manufactured by the microelectronics industry. The proliferation of embedded cores and high-performance computing systems, such as digital signal processing (DSP) circuits, micro-controllers, and micro-processors, makes it possible to use the functionality of these circuits to perform built-in self test rather than adding test hardware which can introduce area overhead and performance degradation. Recently, a new BIST scheme was proposed that utilizes the DSP core to test random logic [37, 38]. The resulting test sessions are controlled by software and use the data-path building blocks, such as adders, multipliers, and ALU, to generate the test patterns and compact the test responses.

5.4.1 Framework

A typical BIST scheme used to test random logic is shown in Fig. 5.6 ([37]).

The scheme uses the STUMPS architecture with the embedded processor serving as pattern generator and signature analyzer. The circuit under test (CUT) features many scan chains that are accessed through a scan buffer (register B) and are governed by two instructions: SCAN.SHIFT and SCAN.LOAD. Instruction SCAN.SHIFT loads the scan

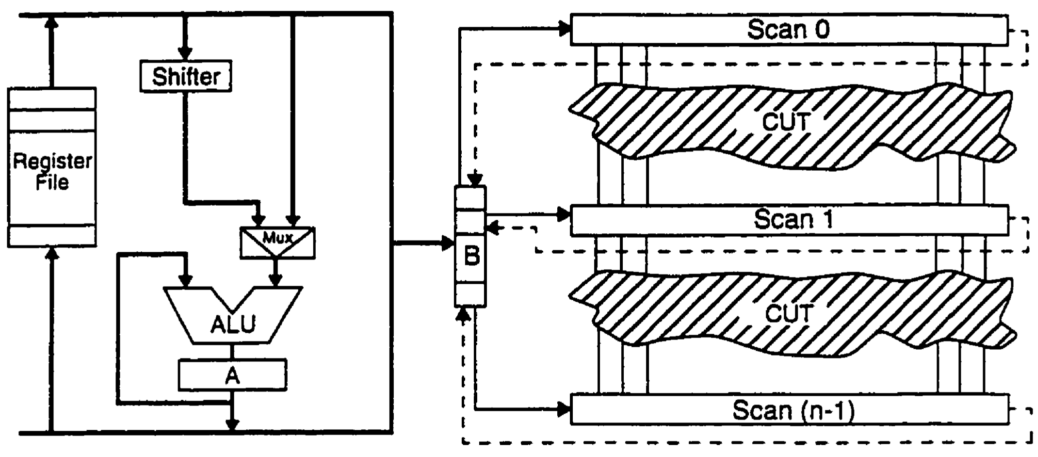


Figure 5.6: CPU core emulating decompressor to test random logic

chains for one cycle and uses register B as input and output of the shift operation. Instruction SCAN.LOAD applies the scan pattern to the circuit under test and loads the response into the scan.

This section demonstrates that the same core can also be used to decompress and apply deterministic patterns. The proposed scheme is a program (or microcode) that reads data from external memory to the local register file and decompresses the data to the scan chains of the circuit under test. Since there is no hardware overhead associated with the scheme, concatenation or variable-length reseeding could equally be used. The proposed scheme uses reseeding based on the concatenation of patterns.

5.4.2 Scheme

One objective of the software scheme is to minimize the number of instructions needed to decompress the data since it has an impact on the test application time. One way to minimize the number of instructions is to exploit the fact that ALUs perform bit wise operations. By using a set of registers (see below), an ALU of width n can execute n LFSR segments in parallel, one segment per bit.

Emulating LFSR-segments in parallel allows us to use one LFSR per scan chain. However, having one LFSR-segment per scan is not sufficient to encode deterministic patterns since the number of specified bits in each scan chain would be limited. In order to encode deterministic patterns, the segments should be linearly interconnected. Such structures were considered previously in [8, 26].

Fig. 5.7 shows an example of linearly interconnected LFSR-segments forming a decompressor of width 3 and length 5. The LFSR-segments are interconnected by adding to their feedback networks taps from other segments. Fig. 5.7 shows the simplest type of connections where each segment is connected to a single neighboring segment and all the taps are taken at the same position. A segment X is said to be connected to segment Y when a tap from segment X is used in the feedback network of segment Y.

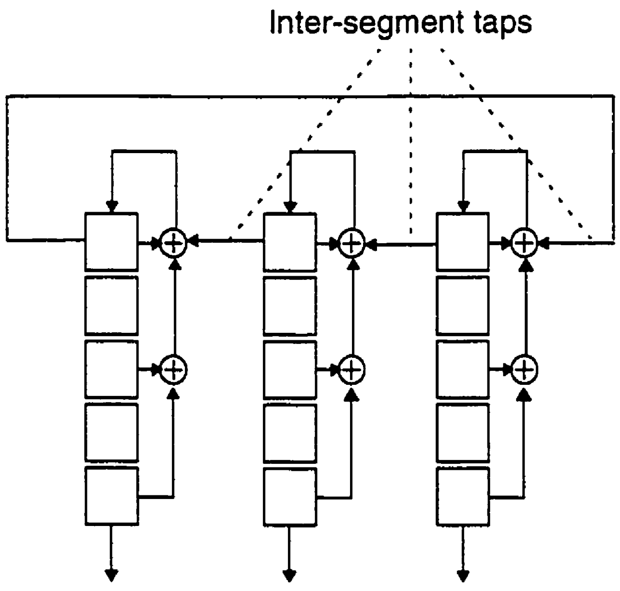


Figure 5.7: Linearly-interconnected LFSR-segments.

Without loss of generality, we assume that the width of the data path, n, is a power of two and is equal to the number of scan chains in the circuit under test. The scheme as presented uses repeated 1's complement addition [38] to perform signature calculations.

Setup

The setup for a decompressor of length L consists of a circular buffer that stores the content of the decompressor and pointer H (Head) used to select the first element of the buffer. The circular buffer is organized such that memory word $M[(H+i) \bmod L]$, $i = 0, 1, \ldots, L-1$ stores the ith bit of each LFSR (Fig. 5.8).

It is assumed that all LFSR-segments implement the same feedback polynomial. This insures that the LFSR-segments can be executed in parallel by the processor. Furthermore, it is assumed that the segments are linked together by an XOR interconnection network as follows: Assuming the LFSRs are numbered from 0 to 2^r-1, the network connects the i^{th} LFSR to LFSRs number $(i+2^v) \bmod 2^r$, for $v = 0, 1, 2, \ldots, r$-1. In addition, the inter-segment taps are taken at the same horizontal position. This insures that each inter-segment connection can be implemented with a combination of one rotate (ROT) and one XOR instruction.

The use of a circular buffer allows for very efficient LFSR shift operations. A shift operation does not involve shifting the content of the memory words: it only involves incrementing pointer H modulo L. As H is incremented, the previously referred memory location can be used to store the new element created by the XOR feedback network. Fig. 5.8 shows the complete framework.

Consequently, one cycle of a two dimensional decompressor, including signature calculation, consists of four main steps: (1) the word $M[H]$ is shifted to the scan chain; (2) the response in register B is added (1's complement) to a reserved memory location (S); (3) XOR and ROT operations are used to calculate the new element generated by the XOR network of the decompressor; and (4) Pointer H is incremented modulo L.

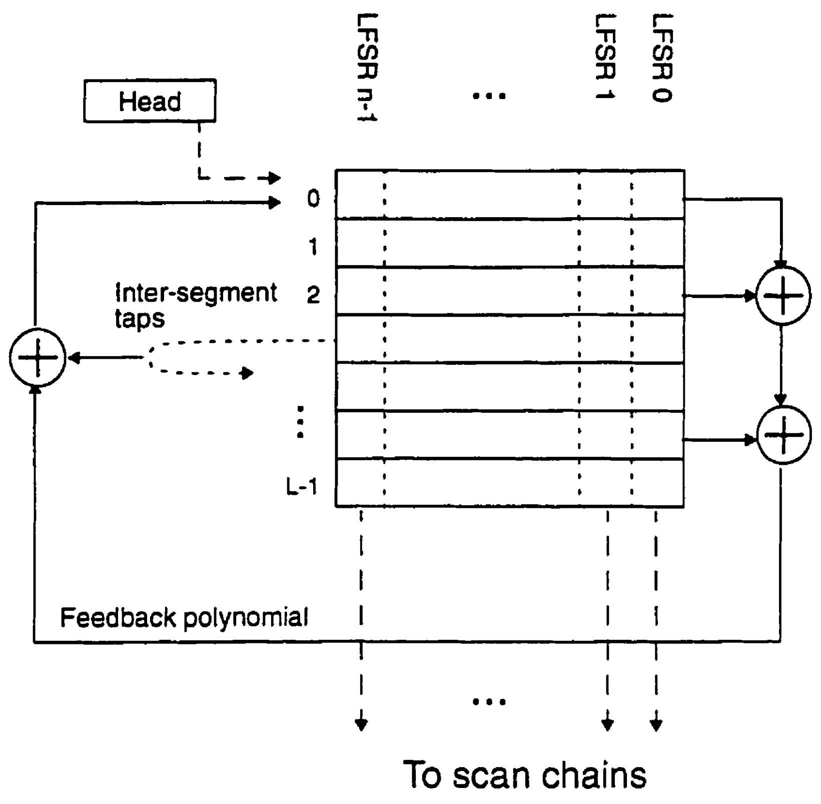


Figure 5.8: Emulation of two dimensional decompressor.

5.4.3 Pseudo-Codes

A pseudo-code implementation of one cycle of the decompressor is shown below.

```
// 1. Shift word M[H] to scan chain
       // Load register B with new element
       B <- M[H];

       // Shift scan chains
       SCAN.SHIFT;

// 2. Signature calculations
       // Add (1s complement) response to location S
       M[S] <- M[S] + B;
       if (OVERFLOW)
          M[S] <- M[S] + 1;
```

```
// 3. Implementation of XOR network
     // For each feedback at position f
     M[H] <- M[H] XOR M[(H+f) mod L];

     // For each inter-LFSR tap at horizontal position T,
     // and relative vertical position v
     M[H] <- M[H] XOR { M[(H+T) mod L] ROT v }

// 4. Increment pointer H
     H <- (H+1) mod L;
```

Loading the seed into the generator merely consists of copying data from one place in memory to another.

Chapter 6 *Experimental Results*

6.1 Introduction

Extensive experiments were conducted to evaluate the merits and trade-offs of variable-length reseeding. The objectives of these experiments were to measure the main parameters of the scheme when applied to "real" circuits and to identify various trade-offs that can be obtained as a function of *test application time*, *area overhead*, *test data*, and *CPU time*. The relationships between these parameters and the number of scan chains were also of particular interest.

The experiments were carried out on the largest ISCAS'89 circuits. The ISCAS'89 benchmarks are a set of 31 digital sequential circuits that were distributed at the International Symposium on Circuits and Systems in 1989 [12]. The circuits are freely available and are used extensively by the research community to evaluate testing schemes. Consequently, results of experiments conducted on ISCAS circuits are often used as reference points.

A mixed-mode test generation scheme was assumed where random and deterministic patterns are used to target all the testable faults in the circuit. The random patterns target the easy-to-test faults. The deterministic patterns cover the remaining hard-to-test faults. The single stuck-at fault model was used for all the experiments. However, transition fault model, multiple stuck-at fault, and other fault models could have been used. The proposed scheme is applicable to all fault models as long as the deterministic patterns can be generated.

6.2 Hardware Implementation

6.2.1 Methodology

The experiment for each circuit consisted of the following steps:

(1) **Insertion of *n* scan chains, each of approximately the same length.** Circuit with 1, 4, 8, 16 and 32 scan chains were considered.

(2) **Fault simulation of 10K random patterns to remove the easy-to-test faults.**

(3) **Generation of deterministic test cubes using ATPG software with dynamic compaction routine.**

(4) **Design of the decompressor (LFSR).** The decompressor was implemented as shown in Fig. 5.3 for circuits with a single scan chains; and as shown in Fig. 5.5 for circuits with multiple scan chains. For the implementation, it was assumed that a 32-bit PRPG (LFSR-type I) was on the chip and could be used. The experiments were repeated for five different PRPGs (P0 = $X^{32}+ X^{29}+X^{11}+ X^{3}+1$; P1 = $X^{32}+ X^{30}+X^{21}+ X^{19}+X^{18}+X^{16}+ X^{14}+X^{5}+1$; P2 = $X^{32}+X^{31}+X^{22}+ X^{20}+X^{14}+ X^{12}+X^{8}+ X^{2}+1$; P3 = $X^{32}+X^{31}+ X^{23}+X^{16}+ X^{14}+X^{11}+ X^{9}+X^{8}+1$; P4 = $X^{32}+X^{28}+ X^{25}+X^{22}+ X^{20} +X^{15}+ X^{13}+X^{2}+ 1$; P5 = $X^{32} + 1$) to show that the results hold for a wide range of PRPG.

For each circuit, the length of the LFSR (used for decompression) was selected such that the test cube with the largest number of specified bits could be encoded with probability 0.999999. From the results of Chapter 4, the length of the LFSR was selected to be

S_{max}+20, where S_{max} is the maximum number of specified bits in a test cube.

(5) **Compression of the deterministic patterns as seed of the LFSR.** As illustrated in Fig. 5.2, the deterministic patterns were encoded as variable-length seeds of the decompressor. The increment d was chosen to minimize the number of extra 0s.

(6) **Computation of the parameters.**

6.2.2 Results

Table 6.1 shows the results after ATPG was performed to generate deterministic patterns targeting hard-to-test faults. The table lists the number of deterministic test cubes (NP), the number of scan flip-flops (NSFF), the maximum and average number of specified bits in a test cube (S_{max} and S_{avg}), the memory requirements to store the deterministic patterns and the CPU time (on a SunSparc 20) necessary to generate the patterns.

Circuit	NP	NSFF	Smax	Savg	Memory Req. (bits)	CPU (sec)
s9234	104	247	112	42	25 688	216
s13207	176	700	183	26	123 200	205
s15850	56	611	249	103	34 216	203
s38417	78	1664	472	204	129 792	389
s38584	52	1464	229	66	76 128	529

Table 6.1: Statistics after deterministic patterns were generated by ATPG

Table 6.2 shows the parameters of the decompressor in each case. The table lists the number of scan chains (NS), the length of the longest scan chain (LS), the number of scan flip-flops re-used to implement the decompressor (SFF), and the number of XOR gates required to construct the feedback network and inter-LFSR taps (NXOR).

The number of XOR gates required depends on the PRPG that was used in the implementation of the decompressor. The table shows the number of XOR gates (NXOR) that would have been required if PRPG P1, P2, P3, P4 or P5 was used instead of PRPG P0.

The results for P1, P2, P3, and P4 were the same. Hence, they were condensed in one column.

As can be seen, the number of scan flip-flops reused is substantial. In some cases, it is almost equal to 500. The column SFF can also be seen as the number of extra flip-flops that would need to be added if scan flip-flops could not be shared. Scheme based on concatenation is an example of a scheme that cannot share flip-flops with the scan chain. Thus, these scheme would require many flip-flops.

Circuit	NS	LS	SFF	NXOR **P0**	NXOR **P1-P4**	NXOR **P5**
s9234	1	247	96	0	0	0
	4	62	96	20	20	20
	8	31	96	24	24	24
	16	16	96	64	48	64
	32	8	96	64	64	96
s13207	1	700	192	0	0	0
	4	175	192	12	12	8
	8	88	192	32	24	24
	16	44	192	48	32	64
	32	22	192	96	64	128
s15850	1	611	256	0	0	0
	4	153	256	12	12	12
	8	77	256	32	24	32
	16	39	256	64	64	64
	32	20	256	128	96	160
s38417	1	1664	480	0	0	0
	4	416	480	36	36	36
	8	208	480	32	32	36
	16	104	480	64	64	64
	32	57	480	160	160	160
s38584	1	1464	224	0	0	0
	4	366	224	32	32	24
	8	183	224	32	24	32
	16	92	224	32	32	64
	32	46	224	64	64	96

Table 6.2: Characteristics of Decompressor for each circuit

Table 6.3 shows the final results after encoding each cube into a variable-length seed. The table lists the number of scan chains (NS), the volume of the compressed test data (TD), and the compression ratio (CR). The volume of compressed data was calculated using the encoding format of Section 5. The compression ratio was calculated by dividing the memory requirements of Table 6.1 with those of Table 6.3.

Circuit	NS	TD (bits)	CR **P0**	Average CR **P1-P4**	CR **P5**
s9234	1	5 346	4.8	5.3	4.7
	4	4 720	5.4		
	8	4 800	5.3		
	16	4 790	5.3		
	32	4 968	5.2		
s13207	1	5 877	21.0	20.1	19.9
	4	5 784	21.3		
	8	5 908	20.8		
	16	5 840	21.0		
	32	6 222	19.8		
s15850	1	6 316	5.4	5.1	4.4
	4	6 269	5.4		
	8	6 286	5.4		
	16	6 320	5.4		
	32	8 612	3.9		
s38417	1	16 797	7.7	7.6	7.6
	4	19 500	6.7		
	8	16 858	7.7		
	16	16 844	7.7		
	32	17 164	7.6		
s38584	1	3 996	19.0	19.2	17.9
	4	3 901	19.5		
	8	3 936	19.3		
	16	3 989	19.0		
	32	4 037	18.8		

Table 6.3: Characteristics of Compressed Data

The compression ratio may depend on the PRPG used in the implementation. The compression ratio obtained with PRPG P0 is compared with the average compression ratio obtained with polynomials P1, P2, P3, and P4; and with the compression ratio obtained with P5. As can be seen, the compression ratio is sensibly the same for each case.

6.2.3 Discussion of Results

Several conclusions can be drawn from Tables 6.1-6.3. Clearly, no additional flip-flops are required to implement the decompression hardware as they are re-used from the scan chain. The volume of compressed data is modest. It is many times smaller than the volume of decompressed data. Also, there is an important trade-off between the number of scan chains, the test application time and the amount of test data: circuits with more scan chains require a shorter application time but need more XOR gates to implement the decompressor and a little more storage.

In all cases, the compression ratio is very impressive. For example, a compression ratio nearly equal to 8 was achieved for s38417. The volume of test data without using any compression technique was 129 792 bits. By using the proposed technique, the volume of test data was reduced to 16 797 bits (for circuit with one scan chain). This is less data than what is required to reload the scan chain eleven times.

If the same compression results obtained with s38417 are applied to the example used in Chapter 1, the memory required to store the test data would be reduced from 54Mbytes to 6.8Mbytes, which would allow less expensive test equipment to test the chip. Furthermore, a board with ten similar chips would require 68Mbytes, instead of 0.5Gbytes.

The most intensive step in terms of CPU usage is test generation. However, it can be seen from Table 6.1 that this step can be done in reasonable time on a workstation. Test generation requires at most 10 minutes for the largest circuit (s38584). Since calculating the seeds is not a computationally intensive task, the CPU times for this step were not reported. For all the circuits, the required time to calculate the seeds was at least an order of magnitude less than to generate the patterns. As an example, 3.4 seconds were required to calculate the seeds for circuit s13207.

6.3 Software Implementation

6.3.1 Methodology

The experimental steps for the software implementation were almost the same as for the hardware implementation:

(1) **Insertion of *n* scan chains, each of approximately the same length**. The experiments were conducted assuming a data path of width 8, 16 or 32. Furthermore, the number of scan chains was assumed to be equal to the width of the data path. Hence, circuits with 8, 16 and 32 scan chains were considered.

(2) **Fault simulation of 10K random patterns to remove the easy-to-test faults.**

(3) **Generation of deterministic test cubes using ATPG software with dynamic compaction routine.**

(4) **Design of the decompressor (LFSR)**. The software decompressor was designed as described in section 5.4. Groups of eight patterns were encoded as fixed-length seeds of the decompressor using the concatenation method that was described in section 4.4.

Thus, for the software implementation, the patterns were encoded using the concatenation technique (see section 4.4). Concatenation was used for the software implementation since the encoding efficiency is very high (as high as for variable-length reseeding) and its major drawback (the necessity to used larger LFSR structures) does not have much impact if the LFSR structure is implemented in software.

For a data path width *n*, the length of the decompressor was chosen such that the size of the decompressor, i.e., $n*L$, is a constant for a given circuit and is greater than $S_{g\text{-}max}$+20, where S_{gmax} is the maximum number of specified bits in a group of eight test cube.

(5) **Compression of the deterministic patterns as seed of the LFSR.**

(6) **Computation of the parameters.**

6.3.2 Results

Table 6.1 shows the characteristics of the decompression structure. The table enumerates, for each circuit, the width of the data path, the polynomial of all the LFSR-segments, the tap positions where the connections between segments are taken, and the list of rotate instructions.

Circuit	Width	Polynomial (Powers of X)	Tap Position	list of Rotates
s9234	8	44, 5, 0	43	1, 2, 4
	16	22, 5, 0	21	1, 2
	32	11, 5, 0	10	1, 2
s13207	8	36, 7, 3, 0	35, 20	1, 2, 4
	16	18, 7, 0	17	1, 2, 4
	32	9, 1, 0	8	1, 2, 4, 8
s15850	8	132, 26, 0	132, 32	1, 2, 4
	16	66, 13, 0	65, 32	1, 2, 4
	32	33, 13, 0	32	1, 2
s38417	8	208, 48, 31, 18, 11, 0	207	1
	16	104, 48, 31, 18, 11, 0	103	1
	32	52, 48, 31, 18, 11, 0	51	1
s38584	8	64, 9, 5, 0	63	1, 2
	16	32, 9, 5, 0	31	1, 2
	32	16, 9, 5, 0	15	1, 2

Table 6.4: Characteristics of software decompressor

Table 6.1 shows the final results. For each circuit, the table lists the number of test groups.

the size of each seed, the storage needed to store the seeds (TD), and the compression ratio (CR). The test data volume is calculated by multiplying the number of test groups by the size of each seed. The table reports the data once for each circuit since the results are independent of the width of the data path.

Circuit	# Test Groups	Size Seed	TD (bits)	CR
s9234	13	352	4 576	5.6
s13207	22	288	7 488	16.4
s15850	7	1056	7 392	4.6
s38417	10	1664	16 640	7.8
s38584	7	512	3 584	21.2

Table 6.5: Final results for software decompressor

6.3.3 Discussion of Results

The degree of the polynomial is equal to the number of memory words need to implement the decompressor. As seen in Table 6.1, for a data path width of 32, all the decompressors can be realized with no more than 52 memory words. Hence, the software decompressor can be implemented with relatively few registers.

The number of terms in the polynomial and the number of rotates give a good approximation of the number of XOR and ROT instructions (per cycle) required to run the decompressor. If the instructions required to handle the circular buffer, performing signature analysis and transferring data to the scan registers are added to the XOR and ROT instructions, the total number of instruction is less than 30. Hence, it is possible to implement the software decompressor in less than 30 machine instructions per cycle.

Chapter 7 *Conclusions*

A novel method has been presented to reduce the amount of test data required to test scan-based designs. The method can be used to design a test procedures that targets all faults but maintains low costs, i. e., low extra area overhead, short test application time, and small amount of test data.

The method reduces the amount of test data by compressing deterministic patterns prior to storing them in memory. During test application, the deterministic patterns are transferred to the chip under test, where they are generated by a decompression unit. Several methods have been presented to design the decompression unit by sharing flip-flops with the scan chain and other structures already on the chip to minimize the area overhead of the scheme.

The technique targets scan-based designs with Signature Analysis or other type of response compaction. It can be applied to external testing to reduce the memory requirements imposed on the external test equipment, or to built-in self test, to the design of an hybrid test pattern generator capable of generating deterministic patterns with modest memory and silicon requirements.

The overall scheme relies on mature ATPG algorithms to generate test cubes for hard-to-test faults with a large proportion of unspecified positions. Dynamic compaction algorithms can be used during ATPG to minimize the number of deterministic patterns to generate. These algorithms target more than one fault with a single pattern.

Whereas all the experiments were conducted with the single-stuck-at fault model, the scheme can be used with any fault model, as long as the deterministic patterns can be generated by ATPG. In essence, the only restrictions of the schemes are:

(1) The deterministic patterns have to be generated by ATPG.

(2) It must be possible to re-order the patterns without affecting the fault coverage. Reordering of the patterns is essential to allow for the seeds to be sorted in increasing order of length. This restriction is met for full-scan and some partial-scan circuits.

(3) The response of the circuit must be strictly binary (0s and 1s) so that it can be compacted into a signature. If the response is not strictly binary, then it cannot be compacted which would minimize the reduction of test data volume...

Finally, several avenues exist for future extensions of the scheme:

(1) Software-based decompression. As the density of packaged chips keeps increasing, it is believed that most of them will contain at least one embedded CPU. In the thesis, it was shown how an embedded core can emulate a wide range of decompression structures with virtually no hardware overhead. However, since embedded cores comprise a fully functional CPU unit, other types of decompression structures could be investigated.

(2) Application to partial-scan designs. Full scan-methods tend to require significant amount of silicon which may be prohibitive. Consequently, alternatives that are based on partial-scan are widely used in the industry. Recall that circuits with partial scan may require a sequence of test patterns to cover a single fault. As presented in the thesis, the proposed scheme cannot encode sequences of patterns and preserve their order. However, extensions could be investigated to apply the technique to partial-scan designs.

References

[1] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*, Computer Science Press, 1990.

[2] Agarwal, and Fung, "Multiple Fault Testing of Large Circuits by Single Fault Test Sets", *IEEE Trans. on Computers*, vol. C-30, no. 11, pp. 855-70, November 1981.

[3] V. K. Agarwal and E. Cerny, "Store and Generate Built-in Testing Approach", *Proc. of FTCS-11*, pp. 35-40, 1981.

[4] V. D. Agrawal, C. R. Kime, and K. K. Saluja, "A Tutorial on Built-In Self Test. Part 1: Principles", *IEEE Design and Test of Computers*, March 1993, pp. 73-82.

[5] V. D. Agrawal, C. R. Kime, and K. K. Saluja, "A Tutorial on Built-In Self Test. Part 2: Applications", *IEEE Design and Test of Computers*, June 1993, pp. 69-77.

[6] H. Anton, *Elementary Linear Algebra*, 6th Edition, John Wiley & Sons, Somerset N. J., 1991

[7] D. B. Armstrong, "A Deductive Method for Simulating Faults in Logic Circuits", *IEEE Trans. on Computers*, C-21, May 1972, pp. 464-71.

[8] P. Bardell, W. H. McAnney, and J. Savir, *Built-In Test for VLSI*, Wiley-Interscience, New York, 1987.

[9] P. H. Bardell, "Calculating the Effects of Linear Dependencies in m-Sequences Used as Test Stimuli", *IEEE Trans. on CAD*, Vol 11, No. 1, January 1992, pp. 83-87.

[10] Z. Barzilai, D. Coppersmith, and A. L. Rosenberg, "Exhaustive Generation of Bit Patterns with Applications to VLSI Self-Testing", IEEE Trans. on Computers, Vol. C-32, No. 2, Feb. 1983, pp. 190-4.

[11] R. W. Bassett et al., "Low-Cost Testing of High-Density Logic Components", *IEEE Design and Test of Computers*, April 1990, pp. 15-28.

[12] F. Brglez, D. Bryan, K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits", *Proc. IEEE Int. Symposium on Circuits and Systems*, May 1989, pp. 1929-34.

[13] C. L. Chen, "Linear Dependencies in Linear Feedback Shift Registers", IEEE Trans. on Computers, Vol. C-35, No. 12, Dec. 1986, pp. 1086-8.

[14] W. T. Cheng, "The BACK Algorithm for Sequential Test Generation", *Proc. International Conference on Computer Design*, 1988, pp. 66-9.

[15] R. D. Elred, "Test Routines Based on Symbolic Logical Statements", *Journal of the ACM*, Vol. 6, pp. 33-6, 1959.

[16] H. Fujiwara, and T. Shimono, "On the Acceleration of Test Generation Algorithms", *IEEE Trans. on Computers*, Vol. C-32, Dec. 1983, pp. 1137-44.

[17] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits", *IEEE Trans. on Computers*, Vol. C-30, No. 3, March 1981, pp. 215-22.

[18] S. W. Golomb, *Shift Register Sequences*, Revised Edition, Aegean Park Press, Laguna Hills CA, 1982.

[19] J. P. Hayes, "Transition Count Testing of Combinational Logic Circuits", *IEEE Trans. on Computers*, Vol. 27, No. 6, June 1976, pp. 613-20.

[20] J. P. Hayes, *Introduction to Digital Logic Design*, Addison-Wesley, New York, 1993.

[21] S. Hellebrand, S. Tarnick, J. Rajski, and B. Courtois, "Generation of Vector Patterns Through Reseeding of Multiple-Polynomial Linear Feedback Shift Registers", Proc. *IEEE International Test Conference*, Baltimore 1992, pp. 120-9.

[22] 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. on Computers*, vol. C-44, Feb. 1995, pp. 223-33.

[23] S. Hellebrand, B. Reeb, S. Tarnick, and H.-J. Wunderlich, "Pattern Generation for a Deterministic BIST Scheme", in *Proc. ICCAD*, November 1995, pp. 88-94.

[24] P. D. Hortensius, R. D. McLeod, and B. W. Podaima, "Cellular Automata Circuits for Built-In Self-Test", *IBM Journal of Research and Development*, Vol. 34, No. 2/3, March/May 1990, pp. 389-405.

[25] D. A. Huffman, "The Synthesis of Sequential Switching Circuits", *Journal of the Franklin Institute*, Vol. 257, pp. 275-303, 1954. Reprint E. F. Moore, ed., pp. 2-62, 1964.

[26] W. J. Hurd, "Efficient Generation of Statistically Good Pseudonoise by Linearly Interconnected Shift Registers", *IEEE Transactions on Computers*, Vol. C-23, 1974, pp. 146-52.

[27] T. P. Kelsey, and K. K. Saluja, "Fast Test Generation for Sequential Circuits", *IEEE Proc. ICCAD*, 1989, pp. 354-7.

[28] B. Koenemann, "LFSR-Coded Test Patterns for Scan Designs", in *Proc. Europ. Test Conf.*, Munich 1991, pp. 237-242.

[29] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes*, Morgan and Kaufmann, San Mateo, 1992.

[30] C. Lin, and S. Reddy, "On Delay Fault Testing in Logic Circuits", *IEEE Trans. on CAD*, Sept. 1987, pp. 649-703.

[31] R. Marlett, "EBT: An Comprehensive Test Generation Technique for Highly Sequential Circuits", *Proc. of Design Automation Conference*, 1978, pp. 332-39.

[32] C. M. Maunder, and R. E. Tullos, *The Test Access Port and Boundary-Scan Architecture*, Los Alamitos, Calif. IEEE Computer Society Press, 1991.

[33] E. J. McCluskey, *Logic Design Principles*, Prentice-Hall, Englewood Cliffs NJ, 1986.

[34] R. Mehio, *Memory-based large LFSRs*, Undergraduate Thesis, Department of Electrical Engineering, McGill University, Montreal, December 1994.

[35] K. C. Y. Mei, "Bridging and Stuck-at Faults", *IEEE Trans. on Computers*, Vol. C-23, pp. 720-7, July 1974.

[36] D. K. Pradhan, and M. Chatterjee, "GLFSR - A New Test Pattern Generator for Built-In Self Test", in *Proc. International Test Conference*, 1994, pp. 481-90.

[37] J. Rajski, and J. Tyszer, "Multiplicative Window Generators of Pseudo-random Test Vectors", *Proc. European Design and Test Conference*, Paris 1996.

[38] J. Rajski and J. Tyszer, "Accumulator-Based Compaction of Test Responses", *IEEE Trans. on Computers*, June 1993, pp. 643-50.

[39] J. Rajski and J. Tyszer, "On Linear Dependencies in Subspaces of LFSR-Generated Sequences", to appear in *IEEE Transactions on Computers*.

[40] J. P. Roth, et al., "Programmed Algorithms to Compute Tests and to Detect and Distinguish between Failures in Logic Circuits", *IEEE Trans. on Electronic Computers*, Vol. EC-16, Oct. 1967, pp. 567-80.

[41] J. Savir, and W. H. McAnney, "A Multiple Seed Linear Feedback Shift Register", *IEEE Trans. on Computers*, Vol. 41, No. 2, Feb. 1992.

[42] D. R. Schertz, and G. Metze, "A New Representation for Faults in Combinational Digital Circuits", *IEEE Trans. on Computers*. Vol. C-21, Aug. 1972, pp. 858-66.

[43] M. Schulz, E. Trishler, and T. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System", *IEEE Trans. on CAD*, Vol 7, Jan. 1988, pp. 125-37.

[44] B. H. Seiss, P. M. Trousborst, M. H. Schulz, "Test Point Insertion for Scan-Based BIST", in *Proc. European Design and Test Conference*, 1991, pp. 253-62.

[45] J. Shen, W. Maly, and F. Ferguson, "Inductive Fault Analysis of MOS Integrated Circuits", *IEEE Design and Test of Computers*, Dec. 1985, pp. 13-26.

[46] G. Smith, "Model for Delay Faults based upon Paths", *Proc. International Test Conf.*, 1985, pp. 342-9.

[47] K. M. Thompson, "Intel and the Myths of Test", *IEEE Design and Test of Computers*, Spring 1996, pp. 79-81.

[48] S. Venkataraman, J. Rajski, S. Hellebrand, and S. Tarnick, "An Efficient BIST Scheme Based on Reseeding of Multiple Polynomial Linear Feedback Shift Registers", *Proc. IEEE/ACM Int. Conf. on CAD*, Santa Clara 1993, pp. 572-7.

[49] R. Wadsack, "Fault Modelling and Logic Simulators of CMOS and MOS Integrated Circuits", *Bell System Technical Journal*, 1978, pp. 1449-73.

[50] J. A. Waicukauski, "A Method for Generating Weighted Random Patterns", *IBM Journal of Research and Development*, March 1989, pp. 149-61.

[51] M. William, and J. Angell, "Enhancing Testability of Large Scale Integrated Circuits via Test Points and Additional Logic", *IEEE Trans. on Computers*, C-22, Jan. 1973, pp. 46-60.

[52] N. Zacharia, J. Rajski, and J. Tyszer, "Decompression of Test Data using Variable-Length Seed LFSRs", *Proc. VLSI Test Symposium*, Princeton 1995, pp. 426-33.

[53] N. Zacharia, J. Rajski, J. Tyszer and J. A. Waicukauski, "Two Dimensional Test Data Decompressor for Multiple Scan Designs", to appear in *Proc. International Test Conference* to be held in October 1996.

Appendix

Obtaining Primitive Polynomials over GF(2) using Maple

There exists numerous tables in books from which primitive polynomials over GF(2) can be obtained. However, it is sometimes necessary to use primitive polynomials with some specific characteristics. For instance, a primitive polynomial of degree 52 with seven terms may be required for an experiment.

Maple is a powerful symbolic math software that allows users to perform all sort of symbolic math operations. Using Maple's built-in functions, it is possible to generate primitive polynomials with user-specific features.

Maple offers many functions that are of particular interests to produce primitive polynomials over GF(2). The most important of these functions is Primitive() mod 2:

- Primitive() mod 2. This function returns true if the polynomial is primitive in GF(2). Otherwise, it returns false.

  Ex: Primitive(x^4+x+1) mod 2; returns true.

By using the above function, a small routine to generate primitive polynomials with user-specified characteristics can be written.

The functions basically produces hundreds and hundreds of randomly generated polynomials according to the user-specified parameters and uses function Primitive() to filter out the non-primitive polynomials.

The routine accepts three arguments: the degree of polynomial, the number of terms, and the number of random polynomials to try. Here is a description of the routine:

```
# This program generates NP random polynomials of degree D mod 2 and
# returns the ones that were found to be primitive mod 2


# NP --> Number of random polys to try
# D  --> Degree of poly
# NT --> Number of terms

F := proc ( D, NT, NP)
 Rnd := rand(1..(D-1)):

 for dummy from 1 by 1 to NP do
  # Make sure that poly has a term of degree D and 0
  q := x^D+1:

  # dummy2 is the number of feedbacks that have to be added
  dummy2 := NT-2:

  # add feedbacks
  while (dummy2 > 0) do
    temp := Rnd():
    if (coeff(q, x, temp) = 0) then
       q := q + x^temp:
       dummy2 := dummy2 - 1:
    fi:
  od:

  # Check if poly is primitive
  if (Primitive(q) mod 2) then
     print(q, 'Primitive'):
  fi:

 od:
  return(`done`):
  end:
```

Let's consider a simple Maple session in which a primitive polynomial of degree 52 with seven terms is generated:

```
     |\^/|      Maple V Release 3 (McGill University)
 ._|\|   |/|_. Copyright (c) 1981-1994 by Waterloo Maple Software and the
  \  MAPLE  /  University of Waterloo. All rights reserved. Maple and Maple V
  <____ ____>   are registered trademarks of Waterloo Maple Software.
       |        Type ? for help.

> F(52,7,100);
                         52         17     26     29     6     22
                        x    + 1 + x    + x    + x    + x   + x   , Primitive
```

Generating Test Cubes with Dynamic Compaction Algorithm

The objective of test cube generation is to detect undetected faults with a minimum number of test patterns. Performing independent test generation on each fault and then attempting to merge the cubes is not effective since it does not consider "merging" when creating the patterns.

Dynamic compaction algorithms consider merging conditions while creating patterns. It proceeds as follows. A fault is selected and test generation is performed. While enforcing all conditions set by the original fault, test generation is attempted on other faults. If successful, then a pattern that targets more than one fault is generated.

The following algorithm, re-copied from [53], was used to perform the experiments reported in Chapter 6.

1. Select a target fault from the current fault list and create a test pattern that detects the fault. Extend the fault detectability of the pattern by attempting to sensitize all faults in the fanout-free network of gates in the fault detection path to any gate in the fault detection path. If a fault is sensitizable to any gate in the fault detection path, it will be highly likely to be detected. Reconvergent fault effects can sometimes result in the fault not being detected.
2. Add the stimulus of this pattern to the test cube and set the internal states of the circuit that result from the stimulus.
3. Randomly select a fault candidate for merging that lies outside the fanout free network of the target fault and any other fault that has been considered for the current test cube. Avoiding faults that share the same fanout free network is desirable since the detectability of many of these faults is already fixed. The fault candidate must also be consistent with the current state of the fault site and must have an unblocked path to an observe point considering the current internal states of the circuit.
4. Perform test generation for the fault candidate with the pattern extension described in step 1. If the test generation is successful, add the stimulus of this pattern to the test cube and set the internal states of the circuit that result from the stimulus.
5. Repeat steps 3-4 until all possible fault candidates are used or until the number of unsuccessful attempts exceed a selected limit (default limit is 200).