Modified Geffe Test Pattern Generator for Built-in Self-test

by

Dandan Qi
B.Sc., University of Victoria, 2003

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Supervisor: Dr. J. C. Muzio

ABSTRACT

Unlike linear Finite State Machines (FSM) such as Linear Feedback Shift Registers (LFSR), the Geffe generator, a nonlinear FSM, hasn't been frequently studied or used in the scenario of digital system testing. Such machines are used as pattern generators for built-in self-test. LFSRs have become widely used in today's integrated circuits since they have a comparatively low hardware overhead. While it is known that a Geffe generator when used as a pattern generator for a built-in self-test, gives improved fault detection, the area overhead is sufficiently high for this not to be a practical approach.

In this thesis, we propose three possible redesigns of the Geffe generator, and these redesigns are analyzed on both theoretical grounds and experiments. Our results show that two of our redesigned machines lead to fault coverage that is comparable to the original Geffe generator, but with very sharply reduced area overhead.

Examiners

Dr. J.C. Muzio, Supervisor (Department of Computer Science)

Dr. M. Serra, Departmental Member (Department of Computer Science)

Dr. H. A. Müller, Departmental Member (Department of Computer Science)

Dr. Kin F. Li, External Member (Department of Electrical & Computer Engineering)

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ALFSM	Autonomous Linear Finite State Machine
ANLFSM	Autonomous Non Linear Finite State Machine
BIST	Built-in Self-test
CLK	Clock
CUT	Circuit Under Test
FSM	Finite State Machine
IC	Integrated Circuit
ISCAS	International Symposium on Circuits & Systems
LCA	Linear Cellular Automata
LFSR	Linear Feedback Shift Register
LHCA	Linear Hybrid Cellular Automata
MUX	Multiplexer
TPG	Test Pattern Generation
VLSI	Very Large-Scale Integration
XOR	Exclusive-or Gate

Chapter 1

Introduction

The focus of this research is the investigation of various structural modifications of conventional Geffe generators, and their applications as Pseudo Random Pattern Generators (PRPG) in Built-In Self Test (BIST).

In BIST, various types of PRPGs are used to generate binary test sequences for evaluating a circuit's fault condition; and each of these generators is implemented by unique design techniques. A PRPG can be a linear Finite State Machine (FSM) such as a Linear Feedback Shift Register (LFSR) or a nonlinear FSM such as a Geffe generator. Unlike linear FSMs, which have been heavily studied in the past [10, 12, 13], nonlinear FSMs have not been extensively studied in the context of digital testing and still require further exploration of their implementation and performance in BIST schemes.

Many linear FSMs are popular choices for test pattern generation in BIST as they are well understood and easy to manipulate. Although very high fault coverage can be achieved when these machines are used to detect stuck-at faults in combination circuits [22, page 1], the research presented by [22] shows a decrease of fault coverage when the testing targets are sequential circuits. This is due to the fact that the test patterns produced by linear machines cannot provide a good variety of pattern transitions [22, page 66].

Nonlinear FSMs can overcome such drawbacks of linear FSMs in sequential circuit testing. In addition, nonlinear FSMs are also used in other research fields, for example, stream-cipher cryptography and delay fault detection of a digital system, and promising results of these researches are received [18, 21]. Therefore, in this research, we study one

of the nonlinear FSMs, the Geffe generator as an important springboard for exploiting nonlinear FSMs for BIST applications.

Similar to other non-linear FSMs, the Geffe generator consists of multiple linear FSMs in its structure. When using a Geffe generator in BIST, we notice that the Geffe generator's capability of producing "good quality" test patterns requires a significant amount of hardware resources from these linear FSMs. This can become a challenge to the BIST design as a requirement of BIST, in addition to high fault coverage, is to keep the overhead of the circuit as low as possible [7].

The above tradeoff of using Geffe generators in BIST reveals a real need for considering a redesign of the conventional Geffe generator, so that the new structure still gives the very high level of fault detection achieved by the original machine, but at a much lower area overhead. This is the fundamental motivation for the proposed modifications to the Geffe machine that are introduced in this thesis.

In our research, we present a set of different modification approaches to further refine the structure of a Geffe generator. The goals of these approaches are to reconfigure the hardware components involved, and to allow the new Geffe generators to achieve maximum efficiency. To introduce our modifications, we outline the main topics, which are covered in the thesis.

In Chapter 2, we provide background materials for this research, including the basic concepts of BIST, the characteristics of linear machines (e.g. LFSR) and the non-linear ones (e.g. Geffe generator), and techniques used in Geffe generator transformation.

In Chapter 3, we introduce the design of our proposed Geffe generator modifications. First, an illustration of the advantages and limitations of using conventional Geffe generators in BIST is discussed. This is followed by a presentation of both the existing and new modifications of Geffe generators. A theoretical analysis of the newly proposed modifications is also included.

In Chapter 4, we discuss the experiment, which are applied to LFSRs, conventional Geffe generators and our modified generators. These experiments are fault simulation of sequential circuits, and the test of the cycle length and transitions of test patterns produced by the various types of generators. The design of our experiments is carefully presented in this chapter.

In Chapter 5, we evaluate the results of the two experiments introduced in Chapter 3. The experimental results of the fault simulation are given and analyzed. Further, we discuss the results from the test of the cycle length and transitions of test patterns used by the simulation. For all the experimental results, comparisons are made among the four categories of generators: LFSR, conventional Geffe generators and two versions of modified Geffe generators. Finally, a conclusion on properties and performance of the new Geffe generators is drawn based on the results and our evaluation.

In Chapter 6, we summarize the main contributions of our research, and we raise several relevant topics for future work.

Chapter 2

Background

The purpose of this chapter is to introduce the relevant background for the later research reported in the thesis. In particular, this includes the following topics: Built-in self-test (BIST), fault modeling, test pattern generation, autonomous linear finite state machines (ALFSMs), autonomous non linear finite state machines (ANLFSMs) and primitive characteristic polynomials. The Geffe generator in its conventional form is introduced in this chapter as a type of non linear finite state machine, since much of the later work is based on various modified Geffe generators, fully explained in Chapter 3.

2.1 Built-in Self-test (BIST)

A digital system is tested and diagnosed in order to check whether faults are introduced during the manufacturing or operation phase. The testing and diagnosis procedure must be quick and effective. A sensible approach for such a requirement is to specify test as one of the system functions; this leads to the topic of built-in self-test (BIST). BIST is the capability of a circuit (chip, board, or system) to test itself [1, page 457]. It is also a technique in which testing (test generation and test application) is accomplished through built-in hardware components [1, page 457]. BIST techniques can be classified into two categories: on-line BIST and off-line BIST. The former refers to the testing that occurs during normal functional operations of a system, and the latter deals with testing a system

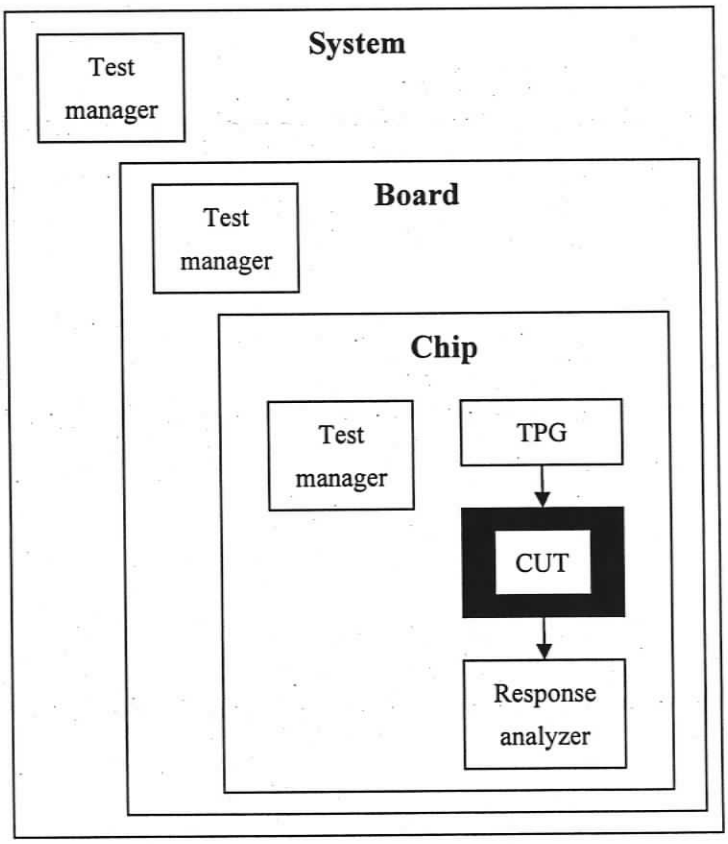
when it is not carrying out its normal functions [1, page 458]. The focus of our research is on the latter.

BIST has the advantages of targeting testing problems such as the complexity issue, the increasing difficulties with test pattern generation, and the economics of testing [2]. As the complexity of Very Large-Scale Integration (VLSI) systems increases, hierarchical procedures are often introduced by engineers to solve the problem of synthesizing and designing a complex system [2]. This raises difficulty in testing the system [2]. The testing requires carrying the test stimulus through many layers of circuitry to the element under test and then returning the result back through many layers of circuitry to the top level [2]. The traditional hierarchical testing techniques, such as deriving a board test from tests for chips on the board, become infeasible and can no longer satisfy the testing needs [2]. However, BIST uses the technique of localizing the testing scope. Therefore, it is an efficient measure for testing systems with such a complexity [2]. Although most savings are at the chip level, BIST offers small savings in testing cost at the system level, as testing costs decrease due to more-efficient tests, less expensive test equipment and improved maintenance testing [2].

The basic BIST architecture requires the addition of three main hardware blocks to a digital circuit: a pattern generator, a response analyzer and a test manager [2]. The pattern generator is responsible for generating a set of input patterns to stimulate the inputs of a circuit, such that detectable faults can be exercised (if present) [1]. Examples of pattern generators are ROM with stored patterns, a counter, and a linear feedback shift register (LFSR) [2]. More details about test pattern generation and various kinds of pattern generators are covered later in this chapter. The purpose of the response analyzer is to check the correctness of the circuit's responses. The test manager is a control block

used to activate the test and analyze the responses [2]. In a hierarchical application of the BIST concept, test-related functions can be embedded at different levels of the system hierarchy [2]. Figure 2.1 shows a typical BIST hierarchy of a system. The test manager at the system level can simultaneously activate self-test on all boards. The test manager on each board, in turn, can activate self-test on each chip on that board. A chip test manager is responsible for executing self-test on the chip and then transmitting the result (fault-free or faulty) to the test manager of the board, which then accumulates test results from all its chips and transmits them to the test manager of the system. Using these results, the system test manager can both detect and isolate faulty chips and boards.

Figure 2.1: BIST hierarchy in a system



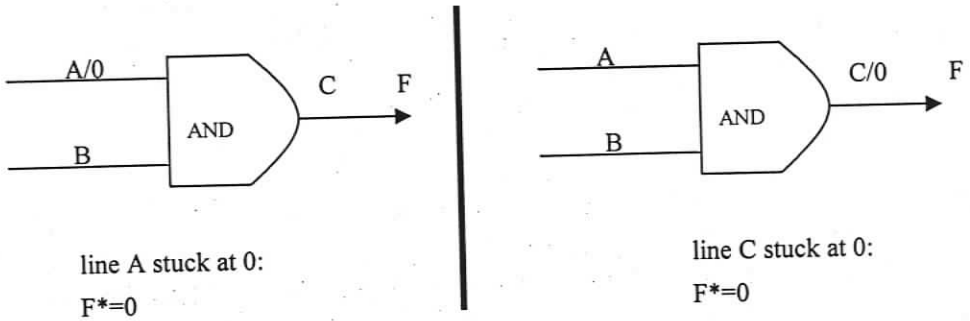
The BIST designs can be defined by the method of test pattern generation (TPG) the BIST employs. Methods in this area include exhaustive testing, pseudo-random testing and pseudo-exhaustive testing [1, page 460]. Different implementations of the TPG have the highest impact on both fault coverage and circuit overheads [7]. The general rule is that BIST TPG must have high coverage and keep the overhead of the circuit low [7]. Throughout this thesis "overhead" refers to the ratio of the extra required hardware for the BIST as compared to the total size of the circuit being tested.

2.2 Logical Faults and Fault Modeling

Many physical mechanisms can contribute to the functional failure of an integrated circuit (IC). These physical mechanisms cause a logical effect and are referred to as logical faults or faults. A fault f is said to be detectable if there exists a test pattern t in a given test set that detects f; otherwise, f is an undetectable fault [1, page 99]. Untestable faults refer to faults whose effect cannot be seen at the output, no matter which test pattern is applied.

Different faults may have exactly the same fault effect (equivalent faults) and need to be covered or simulated only as a set [1, page 107]. For example, in a simple AND gate circuit (Figure 2.2), the fault effects of input stuck at logic 0 and output stuck at logic 0 are the same. For such faulty circuits, the output F* is always logic 0.

Figure 2.2: Example of fault collapsing

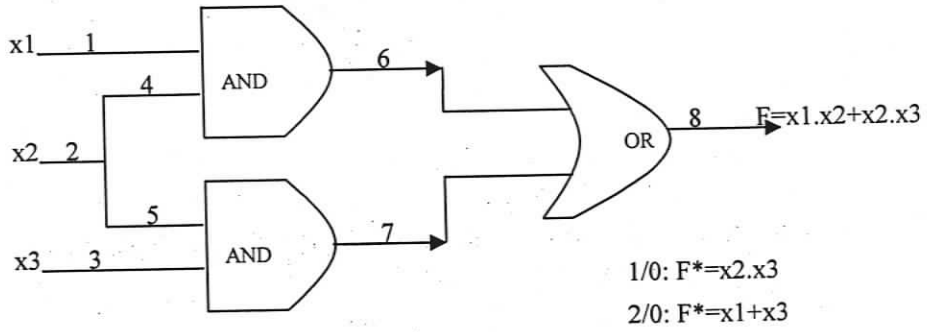


A fault model provides a description of the functional effects of various logical faults [1, page 93]. The construction of logical fault models is the basis of the development of testing algorithms and fault diagnosis methods [1, page 93]. With regards to the logical behavior of fault models, they can be classified, amongst others, into three categories: stuck-at faults, delay faults and bridging faults. Among these three types of fault models, the stuck-at fault model is the most commonly used one [1, page 109]. The quality of a test is most frequently measured as the coverage of single stuck-at faults [2]. Thus, in this thesis only single stuck-at faults are considered.

The stuck-at fault model describes the effect of having a line segment stuck at logical 1 or 0 [1, page 110]. Stuck-at faults can cause a logic gate to behave as though one of its inputs or its output is stuck at logical 1 or 0 [3, page 4]. In order to test whether a stuck-at fault occurs on a line segment, one test pattern is normally required. For example, in Figure 2.3, the circuit has three inputs (x_1, x_2 and x_3) and an output F, where $F= x_1.x_2 + x_2.x_3$. If a stuck-at-0 fault affects line 1 (which is often denoted by 1/0), this circuit's output F is the result of $x_2.x_3$. To test such fault, we can apply test pattern "011" to the

three inputs respectively. If the circuit is fault free, its expected output will be logic 0. However, when the target fault presents on line 1, the output will response with a different output value (logic 1) to the same input test pattern. Similarly, if the circuit contains a single fault: 2/0, it will behave as $x_1 + x_3$. To capture such fault we can use test pattern "111" instead.

Figure 2.3: An example of a stuck-at fault



Single stuck-at faults can be further divided into single stuck-at faults in combinational circuits, and single stuck-at faults in sequential circuits [22, page 6]. For combinational circuits, the single stuck-at faults need only one test pattern to detect as we demonstrate in the above example. However, testing the same type of faults in a sequential circuit is a different scenario. Since the output of a sequential circuit depends not only upon its inputs, but also upon its history, detecting the existence of a single stuck-at fault, therefore, requires more than one test pattern to detect.

2.3 Test Pattern Generation for BIST

As previously mentioned, a set of binary sequences is given to the inputs of a circuit in order to test the presence of faults in the system. These binary sequences are called test patterns or test vectors and Test Pattern Generation (TPG) describes the process of derivation of test patterns [1, page 7]. The set of test patterns for a circuit under test (CUT) is constructed such that the output from the faulty circuit is different from that of the good circuit. Construction of a test requires additional overhead during and after the design stage and it can be extremely time consuming (on the order of 30% of the total development time for an IC) [1, page 7].

A test set can be obtained by two distinct methods: deterministic algorithms and pseudo random methods. The deterministic test pattern generation is computationally expensive; however, the generated test set often leads to good testing results [1, page 226]. In contrast with deterministic test pattern generation, random test pattern generation requires much less computational expense, and it is often implemented in BIST design.

Methods of test pattern generation used in BIST can be summarized into three categories: exhaustive testing, pseudo-random testing and pseudo-exhaustive testing [1, page 460]. We assume that the unit being tested is an n-input, m-output circuit. These different methods are discribed as follows.

- *Exhaustive testing* iterates through all 2^n test patterns, where n is the number of inputs. This test method can detect all detectable faults that do not induce sequential behavior within the circuit. Even with high clock speeds, the time

required to apply the patterns may make exhaustive testing impractical for a circuit with a large number of inputs (e.g. n>25).

- *Pseudo-exhaustive testing* segments a circuit and tests each sub-circuit exhaustively. This approach inherits many benefits of exhaustive testing but lowers the number of test patterns used to achieve satisfied fault coverage. Pseudo-exhaustive testing, however, requires a careful study of the structure of the CUT and a considerable amount of effort must be devoted to circuit segmentation [1, page 462].

- *Pseudo-random testing* generates test patterns that appear to be random but are in fact deterministic (repeatable). Each bit of the test patterns has an approximately equal probability of being 0 or 1. Pseud-orandom testing has the potential for lower hardware and performance overhead and less design effort than the preceding method [1, page 461]. This method can be applied to both combinational and sequential circuits [1, page 461]. Linear Finite State Machines (LFSM) are common pattern generators used by pseudorandom testing.

Compared with the other two test generation methods, pseudorandom testing has several advantages. It has no constraint on the input size of a CUT as exhaustive testing does, and it differs from pseudo-exhaustive testing in that partitioning the CUT is not required. Our research exploits different types of test pattern generators used by pseudorandom testing.

2.4 Autonomous Linear Finite State Machines

Autonomous linear finite state machines (ALFSMs) are widely used in BIST because they are inexpensive in hardware. This makes ALFSMs fairly easy to be integrated with a circuit under test (CUT) for the testing. ALFSMs belong to the category of finite state machines (FSM).

A finite state machine can be described as an algebraic structure of five variables: S, I, Y, M, and δ [19, page 208]. S, I, and Y are finite sets of states, inputs and outputs, respectively, M is an output function mapping from $S \times I$ into S, and δ is a next state function mapping from S into Y. At time t, given the current state s^t, and an input i^t, the next state s^{t+1} is defined by $\delta(s^t, i^t)$:

$$s^{t+1} = \delta(s^t, i^t).$$

The output y^t is obtained by applying function M, which is dependent upon the current state s^t or the current state s^t and input i^t:

$$y^t = M(s^t, i^t).$$

A finite state machine can be in linear or nonlinear in form. If S, I, and Y are each vector spaces over a finite field K, and M and δ are linear functions, the corresponding finite state machine is a linear finite state machine. Its next state s^{t+1} and output y^t can be expressed as [19, page 297]:

$$s^{t+1} = \delta(s^t, i^t) = A \cdot s^t + B \cdot i^t \qquad (2.1)$$

and

$$y^t = M(s^t, i^t) = C \cdot s^t + D \cdot i^t \qquad (2.2)$$

where A, B, C, and D are matrices.

Autonomous linear finite state machines (ALFSMs) are linear finite state machines which have no external input (i.e. i^t =0) so Equation 2.1 and Equation 2.2 become:

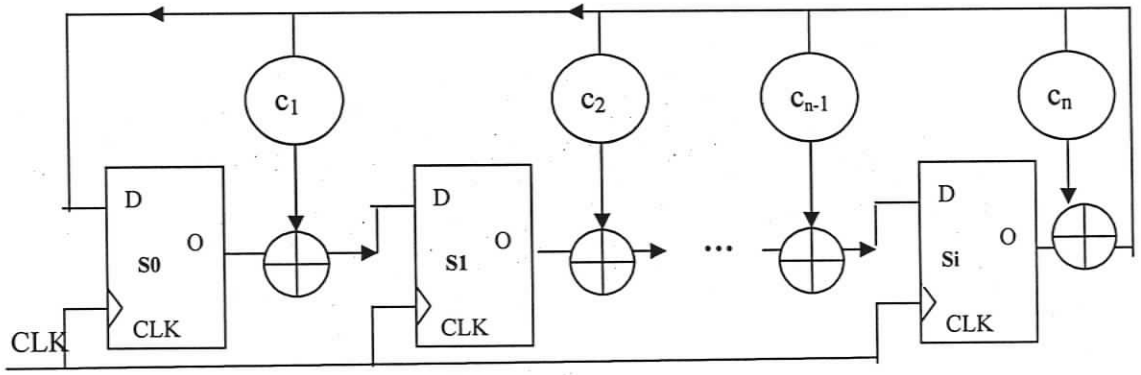
$$s^{t+1} = \delta(s^t, i^t) = A \cdot s^t \tag{2.3}$$

$$y^t = M(s^t, i^t) = C \cdot s^t \tag{2.4}$$

The matrix A and C are the state transition matrix and output matrix respectively.

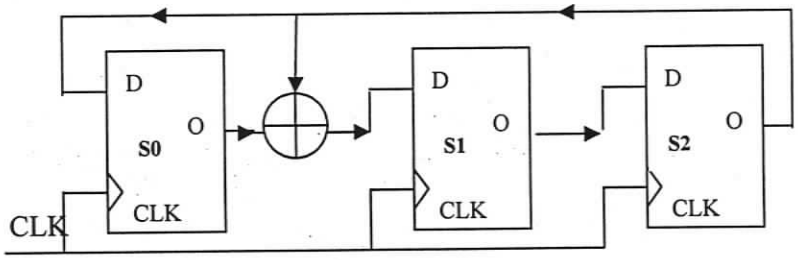
The typical components of an ALFSR are a clock (CLK), memory registers (D flip-flops) and Exclusive-or gates (XOR) which perform addition modulo 2. Memory registers and exclusive-or gates are connected by a network. The network is composed of two types of connections: connections between adjacent registers and connections to feedback taps (XOR gates). In the following, we provide the general structure of an ALFSR (Figure 2.4), and then a 3-bit ALFSR as an example of the ALFSR (Figure 2.5).

Figure 2.4: General structure of ALFSR



Note: c_i is a binary constant, and c_i =1 implies that a connection exists, while c_i =0 implies that no connection exists (i=1,2...,n).

Figure 2.5: An example of ALFSR



By using the Equation 2.3, the next state of the ALFSR in Figure 2.5 is calculated as follows:

$$S_0^{t+1} = S_2^t,$$

$$S_1^{t+1} = S_0^t \oplus S_2^t,$$

$$S_2^{t+1} = S_1^t$$

and in matrix from:

$$S^{t+1} = A \,.\, S^t = \begin{pmatrix} 0\,0\,1 \\ 1\,0\,1 \\ 0\,1\,0 \end{pmatrix} \,.\, S^t$$

so the transition matrix of the ALFSR is

$$A = \begin{pmatrix} 0\,0\,1 \\ 1\,0\,1 \\ 0\,1\,0 \end{pmatrix} .$$

The characteristic polynomial of matrix A is

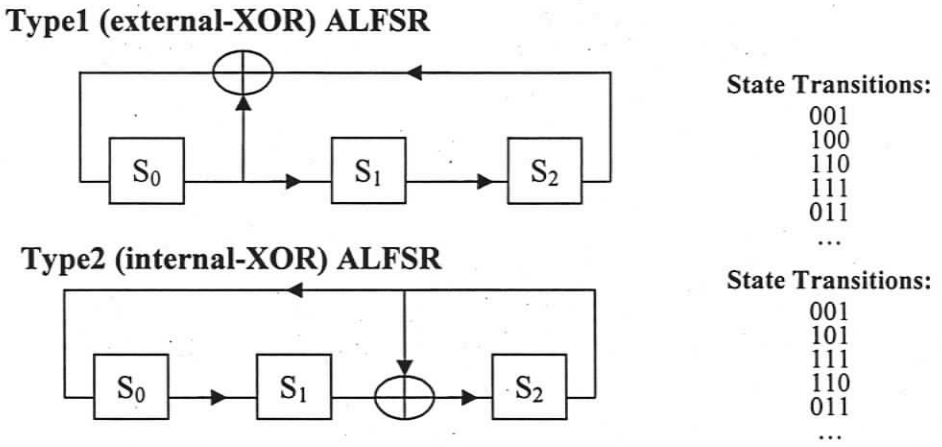
$$f(x) = \det(xI\text{-}A) = \begin{vmatrix} x\,0\,1 \\ 1\,x\,1 \\ 0\,1\,x \end{vmatrix} = x^3 + x + 1.$$

In the above, s_i^t represents the state value of cell i at time t, and s_i^{t+1} represents the state value of cell i at time t+1, $i \in \{0, 1, 2\}$. Symbol $\oplus$ denotes addition modulo 2, det() denotes the determinant of a matrix, and I is the identity matrix [19, page 310].

Since there is a one-to-one correspondence between the ALFSRs and polynomials, to describe a n-cell internal XOR ALFSR, we can use the polynomial $f(x) = c_n x^n + c_{n-1}x^{n-1} + \ldots + c_1 x^1 + 1$, where $c_i \in \{0,1\}$, if c_i=1, 1<i<n, the input of the (i+1)-th cell is the XOR result of the outputs of the i-th and the n-th cells; otherwise, it is the output of the i-th cell. Notice that the next state of ALFSR cannot be the all-zero state (i.e. next state is of the form [0, 0… 0, 0].) if the previous state is not the all-zero state.

There are two ways to position the XOR gates in an ALFSR. Instead of placing them between the cells, we can also arrange them on the feedback line (Figure 2.6) [1, page 434]. Therefore, the ALFSR is classified into two categories: type 1 ALFSR and type 2 ALFSR. Except for the difference in positioning XOR gates, both ALFSR of type 1 and ALFSR of type 2 have similarity in their general structure and state computation. In this research, only ALFSRs of type 2 (internal XOR) are used. For simplicity in the sequel, we use LFSR to refer to an ALFSR of type 2.

Figure 2.6: Type1 ALFSR and Type2 ALFSR



2.5 Non Linear Finite State Machines

Non linear finite state machines are finite state machines, whose next state functions are non linear functions. Among binary operations, linear operation includes only modulo 2 addition, modulo 2 scalar multiplication, and unit delays. Operations such as AND, OR, NAND, and NOR are non linear operations [22, page 13]. Many non linear finite state machines developed in the past are based on applying non linear operations to multiple linear finite state machines [18]. Examples of these machines are the Geffe generator, the Beth-Piper stop-and-go generator and the Massey-Rueppel's generator. Since Geffe generators are the primary subject that we investigate, we discuss the non linear machines with a focus on Geffe generators.

Compared with linear finite state machines (LFSMs), non linear finite state machines adopt different techniques in their design. These design techniques include non linear

feed-forward transformations, step control and multiclock systems [18]. Generally speaking, nonlinear finite state machines have compound structures built according to different design techniques. This property makes nonlinear machines much more difficult to analyze. Nonlinear machines are often expensive to be incorporated into a BIST application.

A conventional Geffe generator is a typical example of a non linear machine. It first described in a popular industry magazine [8], and it comprises three ALFSMs. These three ALFSMs are related to each other by a 2-to-1 mulitplexor (MUX). In this research, ALFSMs involved in constructing a Geffe generator are LFSRs. The general structure of such a Geffe generator is demonstrated in Figure 2.7. According to their purpose, the three LFSRs: $LFSR_0$, $LFSR_1$ and $LFSR_2$ can be classified into two categories: selector LFSR and source LFSR. The $LFSR_0$ is used to provide input for the select bit of the mulitplexor, and it is the selector LFSR. $LFSR_1$ and $LFSR_2$ are source LFSRs. They are the inputs of the MUX. The bit generated by $LFSR_0$ selects either $LFSR_1$ or $LFSR_2$, and the corresponding bit from the selected LFSR is used to join the output stream. The mulitplexor is used to make the selection and creates the nonlinear forward transformation. The output sequence is produced by concatenating these selected bits.

Figure 2.7: The Geffe generator

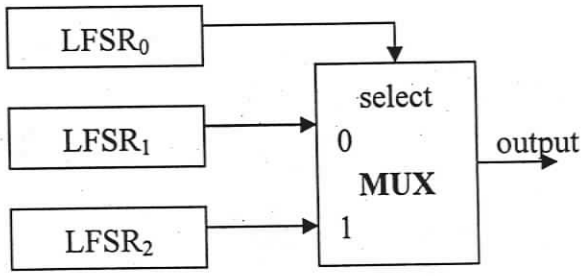
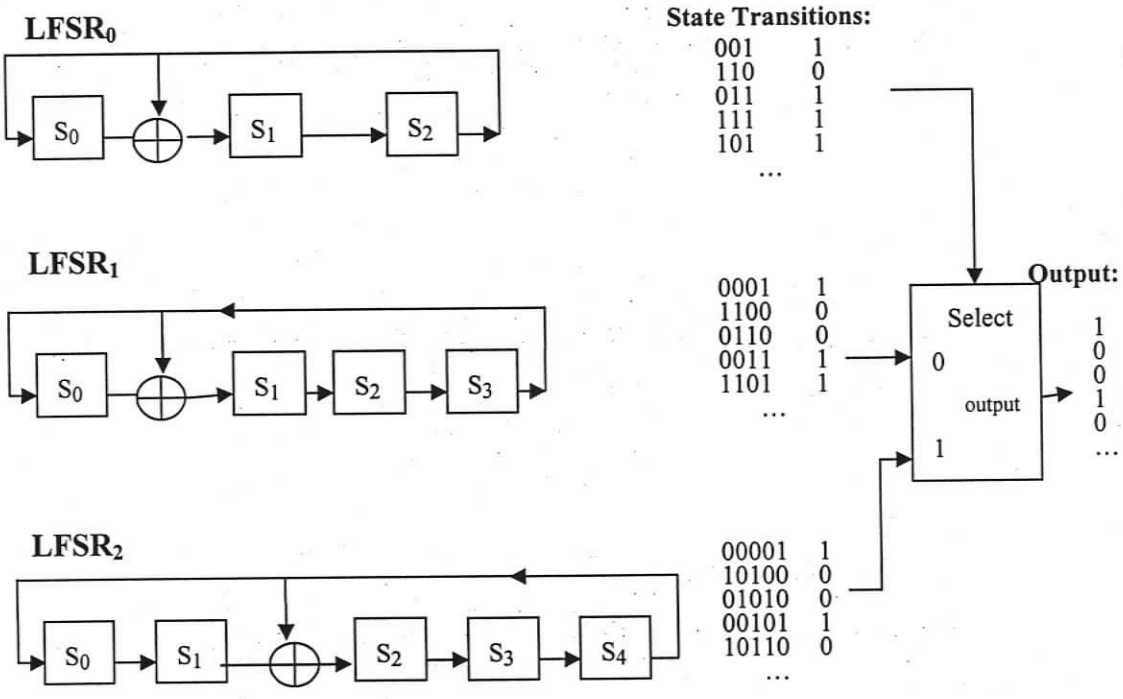


Figure 2.8 shows a more detailed example of the Geffe generator. In the example, all three LFSRs have a starting state which is of the form [0, 0 … 0, 0, 1] (i.e. only the last cell is initialized as '1'). We assume the binary value of the last cell in all three LFSRs is the output bit. The collected output bits from LFSRs are used by the multiplexer to compute the output.

Figure 2.8: An example of the Geffe generator



2.6 Fault Simulation and Fault Coverage

The model of a circuit needs to be exercised by stimulating a set of input signals called a test. The simulation aims to verify the correctness of the model's performance [1, page 3]. When the simulation is applied to the model in the presence of faults, it is the fault simulation.

General fault simulation techniques include serial fault simulation and parallel fault simulation. In our research, we use the latter technique. It allows the fault-free model and a fixed number of faulty models to be simulated simultaneously. HOPE is an efficient parallel fault simulator for the sequential circuits, and it is the primary simulator for our experiment. Detail information if such simulator is discussed in Chapter 4.

Comparing the fault simulation results with those of the fault free simulation, we can determine the faults detected by the given test T [1, page 131]. The ratio of the number of detected faults to the total number of simulated faults can be calculated thereby. This ratio is called fault coverage, and it is an important attribute used to evaluate or grade the given test T [1, page 131]. Therefore, fault simulation is essential to the test generation. Many test generation systems (i.e. generators) use fault simulation as a measure to evaluate the test produced by them. These generators can be compared against each other on their effectiveness of fault detection [1, page 131].

2.7 Primitive Characteristic Polynomial

As we discussed above, the characteristic polynomial for a LFSR $f(x)$ of its $n \times n$ next state matrix A is the polynomial $det(xI - A)$. [19, page 310]. This characteristic polynomial $f(x)$ is of the form:

$$f(x) = c_n x^n + c_{n-1} x^{n-1} + c_{n-2} x^{n-2} + \ldots + c_1 x^1 + 1 \tag{2.5}$$

where $c_i \in \{0, 1\}$ and $0 < i < n$.

2.7.1 Primitivity and LFSR Cycle Length

Fundamental to the primitivity of a polynomial is the concept of irreducible. A polynomial $f(x)$ is irreducible if it cannot be factored by any other polynomial other than 1 and itself [1, page 439]. A primitive polynomial $f(x)$ of degree n is an irreducible polynomial such that $f(x)$ divides x^m+1 evenly for no $m<2^n-1$ [1, page 439].

If a LFSR is constructed by a primitive characteristic polynomial of degree n, it always cycles through all 2^n-1 non-zero states. The output sequence generated by such an LFSR has a period T,

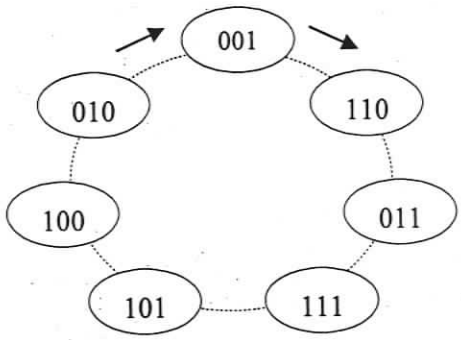
$$T = 2^n-1 \tag{2.6}$$

and this sequence is call a maximum-length sequence [1, page 439]. Figure 2.9 provides an example of such a case. The degree of the characteristic polynomial in this figure is $n=3$, so there are $2^3-1=7$ states that the LFSR can cycle through.

Figure 2.9: Cycle of states

LFSR:
characteristic polynomial: $x^3 + x + 1$

Cycle of States:



If all three LFSRs: $LFSR_0$, $LFSR_1$ and $LFSR_2$ in a Geffe generator are implemented by primitive characteristic polynomials, the cycle length or period T of the Geffe generator is given by equation:

$$T = LCM\,(2^{d0}\text{-}1,\ 2^{d1}\text{-}1,\ 2^{d2}\text{-}1) \qquad (2.7)$$

where d_0, d_1, and d_2 are the degrees of $LFSR_0$, $LFSR_1$ and $LFSR_2$ respectively, and LCM refers to least common multiple. When d_0, d_1, and d_2 are relatively prime to each other, T becomes:

$$T = (2^{d0}\text{-}1) \times (2^{d1}\text{-}1) \times (2^{d2}\text{-}1).$$

2.7.2 Constructing the ALFSM from the Polynomial

Given a binary polynomial, the machines for ALFSRs type1 and type2 can be constructed accordingly. Since the characteristic polynomial and LFSR have a one-to-one relationship, a characteristic polynomial can uniquely define a LFSR [3, page 66]. In practice, we often provide a characteristic polynomial to describe the structure of the corresponding LFSR. The building process is straight forward. Using the Equation 2.5, we first allocate an array of n cells (i.e. memory registers), and number these cell from 0 to (n-1). If $c_i = 1$, $0 < i < n$, the input of the $(i+1)^{th}$ cell is the XOR result of the output of the i^{th} and the n^{th} cells, otherwise, it is the output of the i^{th} cell. The degree of the characteristic polynomial n is also the degree or size of the LFSR.

In Figure 2.5, the LFSR (ALFSRs type 2) shown has the characteristic polynomial $x^3 + x + 1$. The LFSR is constructed by the above rules.

2.7.3 Primitive Characteristic Polynomial Embedding

In this thesis, we specifically propose the idea of characteristic polynomial embedding. It is frequently applied in Chapter 3. The concept of primitive polynomial embedding limits the choices of $p(x)$ and $q(x)$; these two polynomials must be primitive polynomials. For example, polynomial $x^3 + x^2 + 1$ can embedded in polynomial $x^6 + x^4 + x^3 + x^2 + 1$ since we can find $x^3 + x^2 + 1$ in $x^6 + x^4 + x^3 + x^2 + 1$, and both polynomials are primitive polynomials.

If a characteristic polynomial $p(x)$ of degree n can be embedded in another characteristic polynomial $q(x)$ of degree m, it must satisfy the following two conditions:

i) $m \geq n$

ii) $p(x)$ and $q(x)$ must share some common polynomial $f(x)$ such that $f(x) = p(x)$.

Notice that the smallest characteristic polynomial which contains $p(x)$ is $p(x)$ itself. We use $p(x)$ and $q(x)$ to construct two LFSRs, $LFSR_0$ and $LFSR_1$ respectively. If $p(x)$ is embeded in $q(x)$, then we say $LFSR_0$ is contained by $LFSR_1$.

2.8 Summary

Today's integrated circuits (ICs) are in the nature of complex structures and restricted module accessibility. Built-in self-test (BIST) is developed as a cost-efficient testing method mainly because it considerably reduces the expense of test pattern generation and has the ability to apply tests at circuit speeds. A large variety of techniques have been proposed for the generation of pseudo-random test sets for the BIST. The implementation of these test pattern generators (TPGs) requires fairly inexpensive hardware.

Most of the pseudo-random test pattern generators available today are some form of finite state machines (FSM). According to the binary operation of the state computation used by different models of TPG, they can be divided into two categories: linear FSM and non linear FSM. Linear feedback shift register (LFSR) and linear hybrid automats (LHCA) are examples of linear FSMs, while Geffe generator and the Beth Piper stop-and-go generator are examples of non linear FSMs.

The discussion about characteristic polynomials illustrates the idea of primitive characteristic polynomial used to construct LFSRs. A LFSR created from a primitive

characteristic polynomial has a maximum cycle length for its state transition. It is the machine ideal to the circuit testing. If a Geffe generator consists of LFSRs with such properties and the degrees of these LFSRs are co-prime to each other, the output sequence produced by this Geffe generator can also achieve maximum cycle length.

Chapter 3

Geffe Generator Modifications

Chapter 2 discusses various types of FSMs used to generate binary test sequences for evaluating a circuit's fault condition; and each of these machines is implemented by unique design techniques. Unlike linear FSMs, which have been heavily studied in the past [10, 12, 13], nonlinear FSMs have not been extensively studied in the context of digital testing and still require further exploration of their implementation and performance in BIST schemes. The possible reasons for the lack of research on these machines are partially because the nonlinear FSMs could processes the characteristics of nonlinear feed-forward transformations, step control and multiple clocks which introduce great difficulties in the analysis of their behaviors.

In this chapter, one of the nonlinear FSMs, the Geffe generator, is selected as the primary study objective. Compared with other nonlinear FSMs, the Geffe generator has a simpler design with no step control or multiple clocks; therefore it is simpler for detailed analysis. BIST with a Geffe generator as the TPG tends to achieve high fault coverage in the simulation, but with a heavy area overhead penalty [22, page 72]. Both the advantages and constraints of using Geffe generators in BIST are discussed in this chapter. Based on the traditional Geffe structure, three novel and comprehensive techniques for designing cost-effective Geffe generators for BIST are carefully presented. These techniques are

applied to standard Geffe generators, and these modified generators are analyzed separately.

Modification on cellular automata-based Geffe TPGs are not considered in this study as their hardware overhead is higher compared to LFSR based Geffe TPGs and their feedback connections are not represented by simple polynomials, so they are harder to analyze.

3.1 Advantages of Using Geffe Generators in BIST

Besides being widely used in the area of stream cipher cryptography, Geffe generators can also be used in other domains. Geffe generators produce test patterns for fault simulation; and satisfiable fault coverage is often achieved, even in the situation that hard-to-detect faults are present in the circuit under test. This is the major factor which makes Geffe machines as TPG particularly interesting for further studies.

Geffe generators are able to produce test vectors, which give better coverage to detect faults in VLSI testing applications. Binary sequences generated by the Geffe generator provide considerably more transitions than those created by linear machines [22]. This implies a shorter test length or higher fault coverage in the fault simulation of some hard-to-test circuits such as sequential circuits. Moreover, under Maurer's entropy-related statistical test [12, page 27], a test to evaluate the cryptographic strength of a binary sequence, test patterns produced by Geffe generators often achieve a better entropy condition than LFSRs'. According to [18] there is a direct correlation between good entropy as a randomness test and high fault detection. Therefore, Geffe generators used as PRPG for circuits under test can be expected to lead to good fault coverage.

Many experimental results presented in [18, 22] support the correctness of above statements. According to [18], when Geffe machines are used as PRPG in a fault simulation for delay faults, high fault coverage is obtained. Moreover, fault simulation results provided by [22] show Geffe generators have equal ability in detecting stuck-at faults in combinational circuits as linear generators do. In the applications of diagnosing stuck-at faults in sequential circuits, the Geffe generator performs much better than the linear machines.

Because of these benefits, Geffe generators have been suggested as a desirable pseudo-random pattern generators in the arena of obtaining a high coverage for delay faults and stuck-at faults for VLSI.

3.2 Limitations of Using Geffe Generator in BIST

A Geffe generator's capability of producing "good quality" test patterns requires a significant amount of hardware resources. This can become a challenge to the BIST design. The main drawbacks of using Geffe generators in chip testing can be summarized by the following:

- Additional pins and silicon area needed for implementing multiple LFSRs,

  When adopting a Geffe generator in a pseudorandom testing scenario, the generator itself is a complex and nonlinear type of circuit. On the most fundamental level of circuit logic, this means the number of D flip-flops with feedback will be tripled in number, because three LFSRs are involved in the system. When testing an n-input

digital system and n is large, the amount of the hardware resources required by the generator becomes unreasonable. A general rule of thumb for BIST TPG given by [7] is that TPG must have high fault coverage and keep the overhead of the circuit low. Therefore, when taking account of the testing cost, the Geffe generator is not a favorable choice for actual BIST applications even though it is able to achieve high fault coverage [7, page 8].

- Decreased reliability due to the increased silicon area,

The reliability of a system is the probability that a given system operates properly under specified conditions for a specified period of time without failures [14]. By the exponential failure law, the reliability varies exponentially as a function of time and failure rate [14]. In the failure rate calculation, the failure rate partially depends on the complexity factor of an IC such as number of gates and number of flip-flops in the system [14]. In the BIST environment, a Geffe TPG introduces a large number of extra hardware components on the top of the overall testing unit. As the size of the BIST hardware increases, the reliability of the system tends to decrease.

- Performance impact due to the additional circuitry,

The hardware overhead required for BIST also has an impact on the performance of the circuit in normal operation. It is important to ensure that the use of a Geffe generator for BIST does not have a greater impact on the performance of the circuit than the other BIST approaches.

- Additional design time and cost.

Compared with LFSR TPG, Geffe generators have a more complex structure, thus they require additional design effort and a longer period of time to be fully

implemented [20]. Given that over 50% of the design effort on a modern IC is devoted to testing issues, anything which increases the complexity of the design will potentially lead to longer time to market.

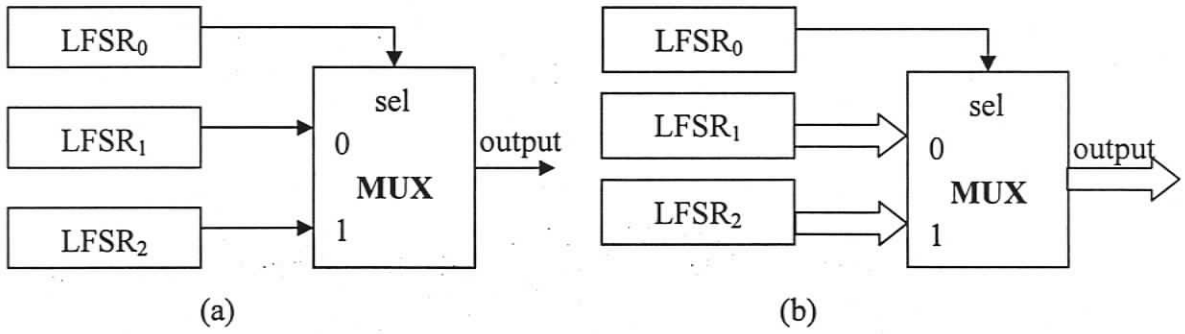
In a quantitative evaluation of a test method, the above concerns about utilizing a Geffe generator in BIST strongly suggest that the traditional implementation of such methodology in circuit designs will raise thorny problems associated with test economics. In the scenario where a Geffe TPG is used to obtain a higher level of fault coverage, it is important to find a balance between a higher fault coverage, and the potential increase in test cost, whether it be in time or area.

Most importantly, the above design tradeoffs reveal a real need for considering the design of new cost-efficient TPG based on the conventional Geffe generator, such that the new generator still gives the very high level of fault detection achieved by the original machine, but at a much lower area overhead. This is the fundamental motivation for proposing modifications to the Geffe TPG in the next section.

3.3 Existing Modifications of Geffe Generators

A slightly different approach to operate a Geffe generator was proposed in [22, page 14]. Figure 3.1 illustrates the structural comparison between the original Geffe generator and the proposed version in [22].

Figure 3.1 The Geffe generator: (a) Original version; (b) Modified version



The modified generator presents a different scheme of generating test patterns. At any given time, the generator outputs a binary test vector that is n bits wide where n is set to be equal to the number of inputs in the CUT. Table 3.1 shows an example for n=3. The bold bit from $LFSR_0$ selects which one of the test vectors produced by $LFSR_1$ and $LFSR_2$ is chosen for the output. Notice that for the modified Geffe generator, the left most n bits taken from $LFSR_1$ and $LFSR_2$ are used for the output, while the original generator uses the right most bit for its output.

Table 3.1 State transitions of a Geffe generator and its modification

Time t_i	$LFSR_0$ x^3+x+1	$LFSR_1$ x^3+x^2+1	$LFSR_2$ x^4+x+1	Output Pattern (a)	Output Pattern (b)
1	00**1**	001	**000**1	1	000
2	11**0**	**101**	1100	1	101
3	01**1**	111	**011**0	0	011
4	11**1**	110	**001**1	1	001
5	10**1**	011	**110**1	1	110
6	10**0**	**100**	1010	0	100

From the above example, we see that the original Geffe TPG takes a full clock cycle to produce a single test bit in the output sequence. However, when a Geffe machine is manipulated according to the above modification, a binary test vector of certain length can be produced in a single clock cycle. In the modified Geffe structure, $LFSR_0$ is used solely to select a test vector instead of a single bit between two other LFSRs at any given time slice t_i. To complete a test set of x binary patterns of length n each, it only requires the improved Geffe machine to run for x clock cycles, if both of $LFSR_1$ and $LFSR_2$ have degree greater or equal than n. However, a conventional Geffe generator may take $x*m$ clock cycles to fulfill the same task. Therefore, the two generators have a significant difference on their run time for the test pattern generation.

Let l, m, and n be the degree of $LFSR_0$, $LFSR_1$, and $LFSR_2$ respectively. The modified Geffe generator can be denoted as Geffe*(l, m, n). In general, the period T of output sequence produced by Geffe*(l, m, n) is calculated to be

$$T= LCM (2^l -1, 2^m -1, 2^n -1),$$

where LCM refers to the least common multiple. If the degrees of the three LFSR's are co-prime to each other, the period T then becomes

$$T= (2^l -1)* (2^m -1)* (2^n -1)$$

The period of test sequences produced by the new machine is exactly the same as the period of the original Geffe generator. [21] claims that if a pattern generator has a longer pattern cycle length, it normally gives a better fault coverage in the circuit. This, thus, implies that the fault detection strength of a Geffe generator, before modification and

after modification, will be very similar. Our experimental results of stuck-at fault simulation for ISCAS 89 benchmark set in [16] also reinforce the accuracy of this hypothesis.

3.4 Structural Modifications of Geffe Generator

The modification described in the previous section on the output scheme of the original Geffe generator reduces the runtime required by the test pattern generation. However, this doesn't address the necessary hardware reduction for the design. As a result, we propose a set of different modification approaches to further refine the structure of a Geffe generator.

A solution to the problem of lowering a generator's physical area is to reconfigure the hardware components involved. In Chapter 2, we show that the basic hardware components of a Geffe generator consist of three LFSRs and one 2-to-1 mulitplexor, but not all of these logic blocks have potential to be modified. The 2-to-1 mulitplexor is such a case. The mulitplexor has a simple logic and performs non linear operations on all three LFSRs; therefore it is not amenable to making changes. Moreover, the area of the mulitplexor is comparatively small, in comparison with the rest of the Geffe generator. However, the implementation of the three LFSRs occupies a large amount of memory space, which provides an optimum opportunity for the overhead reduction. One critical point to be made here is that techniques used to transform the LFSRs in a Geffe generator should not affect the generator's fault coverage in testing.

In this section, we develop three modification approaches to reduce the hardware overhead for Geffe generators. These new approaches also accommodate the goal of allowing the new Geffe generators to achieve maximum efficiency.

3.4.1 Notation Defined

$G(d_0, d_1, d_2)$ denotes the original Geffe generator (Chapter 2, Figure 2.6) used in this section, where

d_0 = the degree of $LFSR_0$,

d_1 = the degree of $LFSR_1$, and

d_2 = the degree of $LFSR_2$.

d_0, d_1 and d_2 are co-prime, and $d_1 < d_2$. $G(d_0, d_1, d_2)$ is used to produce d_1 bit wide test patterns for testing circuits with d_1 inputs. The cycle length of the generated test pattern sequence is denoted as $\boldsymbol{T}$.

$G_1(d_1, d_2)$ denotes the modified machine of $G(d_0, d_1, d_2)$ based on Modification 1. The cycle length of its output is denoted as $\boldsymbol{T_1}$.

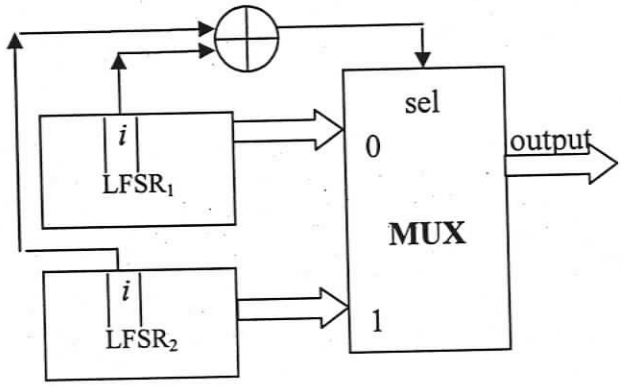
$G_2(d_0, d_3[d_1])$ denotes the modified machine of $G(d_0, d_1, d_2)$ based on Modification 2, where d_3 is the degree of $LFSR_3^+$. The cycle length of its output is denoted as $\boldsymbol{T_2}$.

$G_3(d_4[d_1])$ denotes the modified machine of $G(d_0, d_1, d_2)$ based on Modification 3, where d_4 is the degree of $LFSR_4^+$. The cycle length of its output is denoted as $\boldsymbol{T_3}$.

3.4.2 Modification 1: Removal of the selector LFSR

Reducing the hardware overhead by eliminating the selector $LFSR_0$ is a simple and intuitive technique to reconfigure a Geffe structure. The selector $LFSR_0$ is replaced by applying XOR to a certain bit/bits taking directly from $LFSR_1$ and $LFSR_2$. We denote the modified generator of $G(d_0, d_1, d_2)$ as $G_1(d_1, d_2)$. Figure 3.2 shows the structure of a Geffe generator after the modification.

Figure 3.2 Geffe generator modification 1



Let $S_{k,i}$ refer to the i^{th} flip-flop in $LFSR_k$, where k=0, 1, 2 and $0 \leq i < d_1$. We use $S_{1,i}$ and $S_{2,i}$ to make the selection. At any given time, if the value of $S_{1,i}$ is not equal to the value of $S_{2,i}$, the left most m bits generated by $LFSR_2$ become a test pattern to join the output stream. Similarly, when $S_{1,i}$ and $S_{2,i}$ have the same logic value, the left most d_1 bits generated by $LFSR_1$ become a test pattern to join the output stream . Table 3.2 gives the first six state transitions of the generator G1(*3, 4*) taking i equal to 0. The i^{th} bit of $LFSR_1$ and $LFSR_2$ is underlined. The selected test vector is shown in bold according to the XOR of $S_{1,i}$ and $S_{2,i}$.

Table 3.2 State transitions of $G_1(3, 4)$

Time t_i	$LFSR_1$ x^3+x^2+1	$LFSR_2$ x^4+x+1	XOR the i^{th} bits of $LFSR_1$, $LFSR_2$ $i=0$	Output Pattern
t1	**001**	0001	0	001
t2	**101**	1100	0	101
t3	111	**0110**	1	011
t4	110	**0011**	1	001
t5	011	**1101**	1	110
t6	**100**	1010	0	100

3.4.3 Modification 2: Modification of the two source LFSRs

Figure 3.3 Geffe generator modification 2

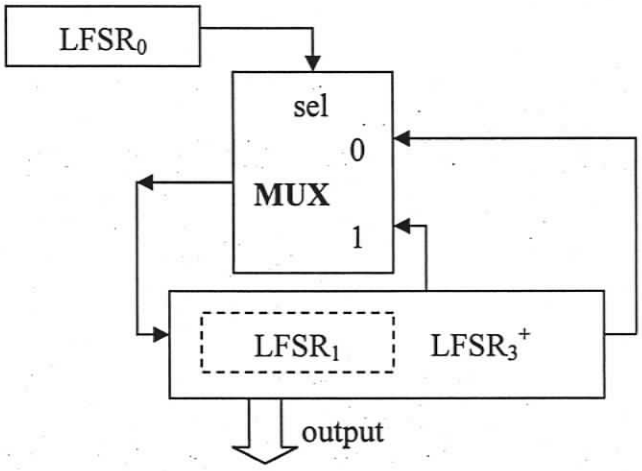
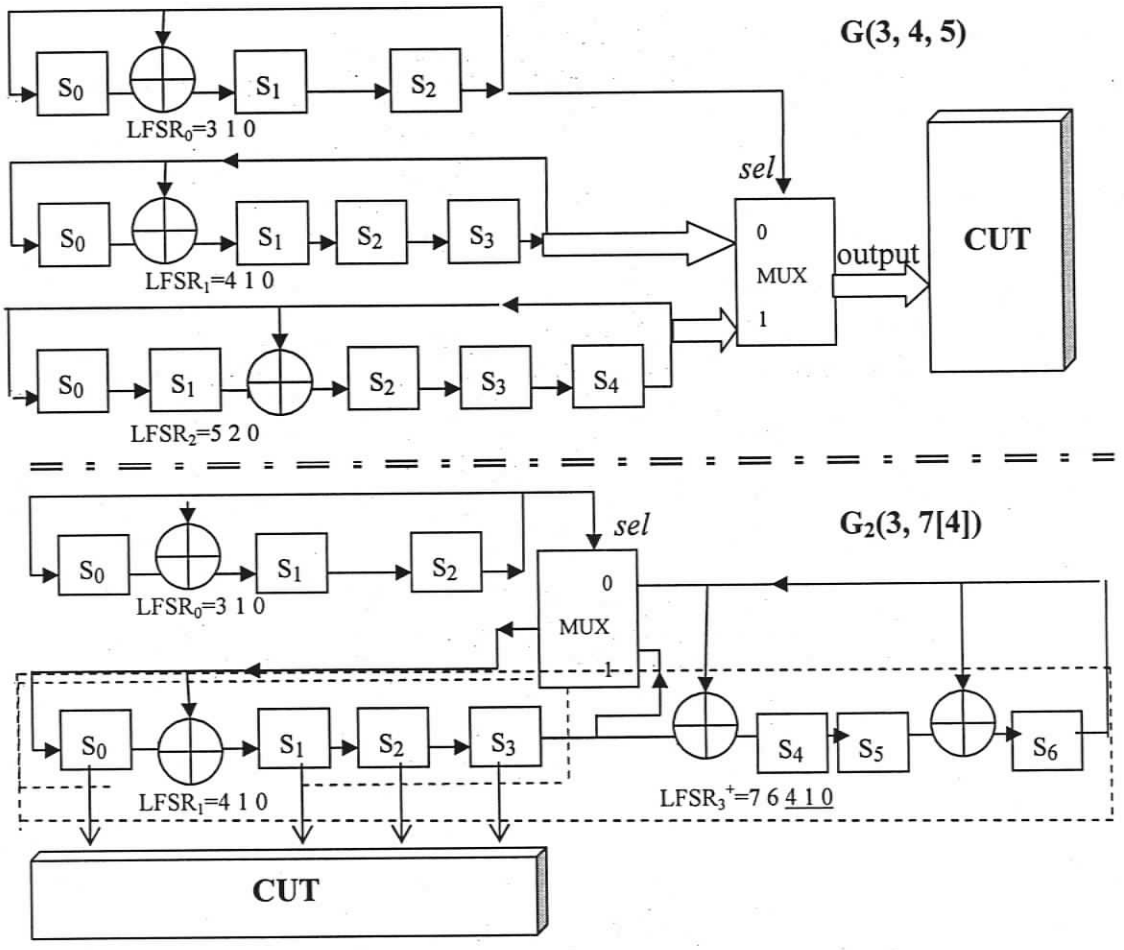


Figure 3.3 shows another possible modification of the original Geffe generator $G(d_0, d_1, d_2)$. We denote such modified machine as $G_2(d_0, d_3[d_1])$. Our attempt here is to embed $LFSR_1$ and $LFSR_2$ into a single LFSR, whose total length is less then d_1+d_2. This

synthesis LFSR is called $LFSR_3^+$. This technique reduces the hardware overhead by cutting down the total number of D flip-flops used in $LFSR_1$ and $LFSR_2$. Note, however, that it is important to ensure that the two embedded LFSRs are still primitive. Figure 3.4 provides an example of the transformation from a original Geffe generator, G(3, 4, 5), to a modified generator, $G_2(3, 7[4])$.

Figure 3.4 Transformation of $G_2(3, 7[4])$



The above transformation procedure consists of the following steps:

- Step 1: Embed $LFSR_1$.

Construct $LFSR_3^+$ of length d_3 with a characteristic polynomial p_3, which satisfies the following requirements:

i) p_3 is a primitive polynomial of degree d_3.

ii) $d_3 \leq d_1 + d_2$, and d_3 is relatively prime to d_0 and d_1.

iii) p_3 contains the characteristic polynomial of $LFSR_1$.

The example in Figure 3.4 shows the three LFSRs: $LFSR_0$, $LFSR_1$ and $LFSR_2$ in the original Geffe machine which have characteristic polynomials of x^3+x+1, x^4+x+1, and $x^5+ x^2+1$ respectively. Polynomial p_3 is determined to be $x^7+ x^6+ x^4+x+1$. This is because p_3 is primitive, and its degree $d_3 = 7 < d_1 + d_2 = 4+5$. Moreover, d_3 is co-prime to the degree of $LFSR_0$ ($d_0=3$) and the degree of $LFSR_1$ ($d_1=4$). The characteristic polynomial of $LFSR_1$ (x^4+x+1) appears in p_3, and $LFSR_1$ is said to be embedded in $LFSR_3^+$.

If more than one polynomial of degree d_3 meets the above requirements for p_3, p_3 is chosen at random. If no such polynomial is found, the original generator can not be modified by such a method. The pseudo code for determining p_3 is given as follows:

```
Boolean Procedure GetPolynomial( Set U, Integer d0,  Integer d1, Integer d2);
    U := a set of all primitive polynomials
    d0:= the degree of LFSR0
    d1 := the degree of LFSR1
    d2 := the degree of LFSR2
    begin
      d := d1+1;
      while( d <= d1+d2 ) do begin
```

```
    P:= a subset of U such that each polynomial p_i in the set has degree d.
    for each p3 ∈ P do begin
       if( p3 satisfies requirement ii) & p3 satisfies requirement iii) )
          return (true);
    end;
    d:=d+1;
 end;
 return (false);
end;
```

- Step 2: Reconfigure the feedback network of $LFSR_3^+$.

Once the $LFSR_3^+$ is obtained, a break point in the feedback network in $LFSR_3^+$ can be determined. The break point is located at the feedback line immediately after the S_j stage, where j is equal to d_1-1. The break point separates the feedback network into two disconnected parts: a left network and a right network.

In the example given in Figure 3.4, the break point can be found at the feedback line immediately after S_3. The original feedback network is disconnected at this point.

- Step 3: Insert the 2-to-1 multiplexer.

The 2-to-1 multiplexer is placed at the previously computed break point. The output of the multiplexer connects to the left feedback network; while the right feedback network connects to one of multiplexer inputs. The other multiplexer input connects directly to the S_j stage.

Figure 3.4 shows the multiplexer is inserted into the feedback network at the break point. The output of S_3 and S_6 are fed back to each of the two multiplexer inputs. In order to reestablish the feedback connection for $LFSR_3^+$, he output of the multiplexer is connected to the input of S_0.

- Step 4: Establish the selector bit for the multiplexer.

The $LFSR_0$ in the original Geffe machine remains the same and the last bit of $LFSR_0$ is the selector bit of the multiplexer. In the new Geffe generator $G_2(3, 7[4])$, the output of S_2 in $LFSR_0$ is connected to the selector input of the multiplexer.

- Step 5: Generate outputs

At any given time t_i, the left most d_1 bits from $LFSR_3^+$ become the test vector for the CUT at time t_i as shown in Figure 3.4 . If the $LFSR_0$ and $LFSR_3^+$ are given starting states of [0,0,1] and [0,0,0,0,0,0,1] respectively, the first five output patterns collected by $G_2(3, 7[4])$ are shown in Table 3.3.

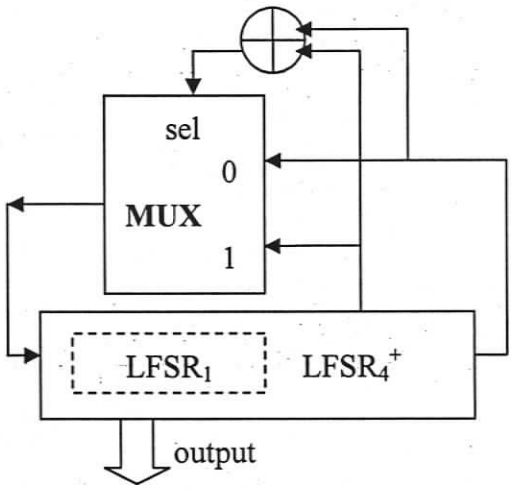
Table 3.3 State transitions of $G_2(3, 7[4])$

Time	$LFSR_0$ x^3+x+1	$LFSR_3^+$ $x^7+x^6+x^4+x+1$	Output pattern
t1	001	**0000**001	0000
t2	110	**0000**101	0000
t3	011	**1100**111	1100
t4	111	**0110**110	0110
t5	101	**0011**011	0011

A new Geffe generator can be obtained by applying the above procedure to the traditional Geffe generator. For such a modified machine, at a time t_i, if its $LFSR_0$ has value 1, its output is the current state of the $LFSR_1$. If the $LFSR_0$ has value 0 at t_i, the output produced by the modified Geffe generator is the first d_1 bits of the current state of the $LFSR_3^+$. Therefore, depending on the output of $LFSR_0$, the generator cycles through some finite states of $LFSR_1$ or $LFSR_3^+$.

3.4.4 Modification 3: Combine three LFSRs

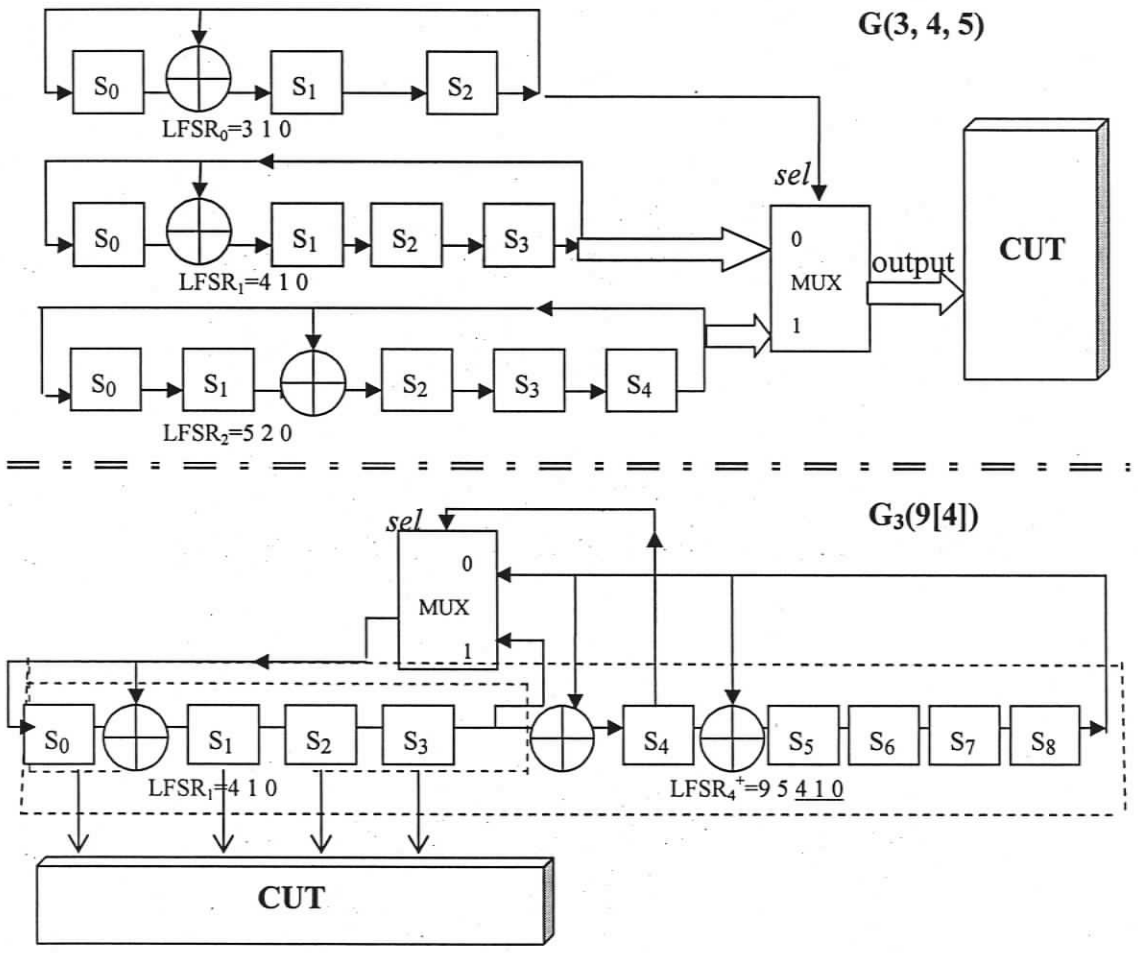
Figure 3.5 Geffe generator modification 3



In this modification, our target model $G_3(d_4[d_1])$ is demonstrated in Figure 3.5. The three LFSRs: $LFSR_0$, $LFSR_1$ and $LFSR_2$ in the original Geffe generator are further integrated into one homogenous LFSR, namely $LFSR_4^+$. Similar modification 2, the integration of three LFSRs also starts with LFSR embedding. In this modification, our attempt is to integrate all three LFSRs into a single LFSR named $LFSR_4^+$, whose total length is less than $d_0+d_1+d_2$. The degree of $LFSR_4^+$, however, should not be too small (otherwise the cycle length of the modified machine is small). We consider d_0+d_1 to be a reasonable lower bound for the degree of $LFSR_4^+$. Since the length of $LFSR_4^+$ is less than the sum of the degrees of the original LFSRs have, the number of flip-flips of this new structure can be reduced in this way.

Observation from Figure 3.5 shows that this structural transformation shares many similarities with Modification 2. Both approaches use the technique of LFSR embedding, and their output schemes are very similar. Therefore, it is easier to build $G_3(d_4[d_1])$ based on Modification 2. In the following figure, we provide a concrete example to demonstrate the building process.

Figure 3.6 Transformation of $G_3(9[4])$



- Step 1: Integrate three LFSRs.

In Modification 2, we construct $LFSR_3^+$ with a characteristic polynomial p_3. For this modification, we use a similar approach to build $LFSR_4^+$. Construct $LFSR_4^+$ with a characteristic polynomial p_4; however, the requirements of p_4 must be revised as follows:

i) p_4 is a primitive polynomial of degree d_4.

ii) d_4 is relatively prime to d_1, and $(d_0+d_1) < d_4 \leq (d_0+d_1+d_2)$.

iii) p_4 contains the characteristic polynomial of $LFSR_1$.

Providing these requirements to the Procedure GetPolynomial introduced in Modification 2, we determine p_4 for the $LFSR_4^+$ in $G_3(9[4])$ is $x^9+x^5+x^4+x+1$. Such p_4 is primitive, and its degree $d_4=9$, $d_0+d_1 = 7 < d_4 < d_0+d_1+d_2 = 12$. Moreover, d_4 is co-prime to the degree of $LFSR_1$ ($d_1=4$). Most importantly, the characteristic polynomial of $LFSR_1$ (x^4+x+1) appears in p_4, and $LFSR_1$ is said to be embedding in $LFSR_4^+$.

If more than one polynomials of degree d_4 meet the above requirements for p_4, p_4 is chosen at random. If no such polynomial is found, the original generator can not be modified by such a method.

- Step 2 and Step 3: same as the Step 2 and Step 3 in Modification 2.

- Step 4: Establish the selector bit for the multiplexer.

The value of S_{d1} is the bit acting as the selector of the multiplexer, where d_1 is equal to the degree of the characteristic polynomial of $LFSR_1$. S_{d1}, is the XOR value of the two inputs of the multiplexer. In the example given in Figure 3.6, the selector bit is directly taken from S_4.

- Step 5: Generate outputs

At each clock cycle, the bit values of the first d_1 bits of $LFSR_4^+$ are grouped and they are a test vector for the CUT. If $LFSR_4^+$ in Figure 3.6 is given starting state of [0, 0, 0, 0, 0, 0, 0, 0, 1]. The first five output patterns collected by $G_3(9[4])$ are shown in Table 3.4.

Table 3.4 State transitions of $G_3(9[4])$

Time	$LFSR_4^+$ $x^9+x^5+x^4+x+1$	Output pattern
t1	**000**00001	0000
t2	**110**11000	1100
t3	**011**01100	0110
t4	**001**00110	0011
t5	**000**10011	0001

The above illustrates a way to modify a Geffe generator so that its overhead can be reduced. When the cells S_{d1-1} and S_{d4-1} in the modified generator have different logic values, the generator will cycle through some finite state of the $LFSR_1$. Similarly, when the two cells have the same logic value, modified Geffe generator will cycle through some finite state of the $LFSR_4^+$.

3.5 Analysis of the Modified Geffe Generators

Three possible modifications to Geffe generators are introduced in the previous section. In this section, we analyze these potential solutions. Our analysis focuses on the study of

the flip-flop reduction resulting from the modifications and their effect on the generators' fault detection capabilities.

At the start of the analysis, our notation is restated as it is frequently referred to through out this section.

1) $G(d_0, d_1, d_2)$: it denotes to an original Geffe generator, whose three LFSRs: $LFSR_0$, $LFSR_1$ and $LFSR_2$ have degree d_0, d_1 and d_2 respectively. d_0, d_1 and d_2 are co-prime to each other. This generator will be redesigned by three modification approaches suggested in the last section.
2) Number of flip-flops in $G(d_0, d_1, d_2)$: there are $(d_0 + d_1 + d_2)$ flip-flops used by the three LFSRs in the generator.
3) T: it refers to the cycle length of the test sequences generated by $G(d_0, d_1, d_2)$. T equals to $(2^{d0} -1) \times (2^{d1} -1) \times (2^{d2} -1)$ by Equation 2.7 given in Chapter 2.

3.5.1 Analysis of Modification 1

Flip Flop Reduction

Modification 1 lowers the hardware overhead by eliminating the selector LFSR in a Geffe machine. After the modification, for the new machine $G_1(d_1, d_2)$, the reduction in the number of flip-flops is:

Number of flip-flops in $G_1(d_1, d_2) = d_1 + d_2$

Number of flip-flop reduction = d_0

$$\text{The percentage of reduction} = \frac{d_0}{d_0 + d_1 + d_2} \times 100\%$$

From the calculation above, the reduction in the number of flip-flops depends on the degree of the characteristic polynomial of the selector LFSR, $LFSR_0$. The larger the size of $LFSR_0$, the greater the reduction in hardware required to build a Geffe generator. However, normally the degree of $LFSR_0$ is much less than the degree of the other two LFSRs, so the saving is not as significant as might be hoped.

Prediction on Fault Detection Performance

As suggested in [21], measuring the period of the binary sequenced generated by a TPG usually provides a good prediction on how well the TPG will perform in testing applications. According to Modification 1, the d_0 degree selector $LFSR_0$ is replaced by the XOR of some stages from the two source LFSRs. The modified structure will create a sequence of selector bits such that the sequence has a cycle length T^*. T^* is given by

$$T^* = \text{LCM}((2^{d1} -1), (2^{d2} -1))$$

where d_1 and d_2 refer to the degree of the polynomial of the two source LFSRs in the original Geffe generator. Therefore, the period of output patterns generated by the new Geffe model will have cycle length T_1

$$T_1 = \text{LCM}(T^*, (2^{d1} -1), (2^{d2} -1)) = \text{LCM}((2^{d1} -1), (2^{d2} -1))$$

If d_1, and d_2 are co-prime to each other, this cycle length T_1 becomes

$$T_1 = (2^{d1} - 1) \times (2^{d2} - 1) .$$

Since T_1 is much less than T, this would seem likely to lead to deterioration in the resulting fault coverage. Given our earlier comment that the potential saving of this approach are not very high, this modification is not a promising solution to the problem of area overhead penalty introduced by Geffe generators.

3.5.2 Analysis of Modification 2

Flip Flop Reduction

Modification 2 lowers the hardware overhead by combining $LFSR_1$ and $LFSR_2$ from the conventional Geffe machine. After the modification, for the new machine $G_2(d_0, d_3[d_1])$, the number of flip-flops reduced can be calculated as follows:

Number of flip-flops in $G_2(d_0, d_3[d_1]) = d_0 + d_3$

Number of flip-flop reduction = $(d_1 + d_2) - d_3$

The percentage of reduction = $\dfrac{(d_1 + d_2) - d_3}{d_0 + d_1 + d_2} \times 100\%$

By the calculation above, the reduction in the number of flip-flops depends on the degree of characteristic polynomial used to build $LFSR_3^+$. Modification 2 attempts to construct one slightly larger LFSR to replace the $LFSR_1$ and $LFSR_2$ in the original machine. The degree of $LFSR_3^+$ can not be too small otherwise the $LFSR_1$ embedding will not succeed,

and deterioration in the resulting fault coverage will occur. Our research discovers that a reasonable degree for $LFSR_3^+$ is approximately d_1+2. To test a CUT with n inputs, we usually set the original Geffe generator as its $LFSR_1$ and $LFSR_2$ to have degree n and n+1 respectively. For the same testing scenario, the original Geffe generator is modified using approach 2. When n is small, $LFSR_1$ and $LFSR_2$ are replaced by some suitable $LFSR_3^+$, but the flip-flip reduction is not obvious as shown in Table 3.5. However, when n grows larger, the reduction in the number of flip-flops will become more and more significant. This is because the degree of $LFSR_2$, d_2 becomes much larger than d_1+2, which is the degree approximation of $LFSR_3^+$.

Table 3.5 Characteristic polynomials used by Modification 2

# inputs in CUT n	$G(d_0, d_1, d_2)$			$G_2(d_0, d_3[d_1])$		# FF reduced
	$LFSR_0$	$LFSR_1$	$LFSR_2$	$LFSR_0$	$LFSR_3^+$	
3	x^5+x^2+1	x^3+x^2+1	x^4+x+1	x^5+x^2+1	$x^7+x^4+x^3+x^2+1$	0
4	x^5+x^2+1	x^4+x+1	x^5+x^2+1	x^5+x^2+1	$x^7+x^6+x^4+x+1$	2
7	x^5+x^2+1	$x^7+x^6+x^3+x+1$	$x^8+x^5+x^3+x+1$	x^5+x^2+1	$x^9+x^8+x^7+x^6+x^3+x+1$	6
9	x^5+x^2+1	x^9+x^5+1	$x^{10}+x^3+1$	x^5+x^2+1	$x^{11}+x^{10}+x^9+x^5+1$	8

[Assume x^5+x^2+1 is the characteristic polynomial of $LFSR_0$ for all Geffe machines.]

Prediction on Fault Detection Performance

According to Modification 2, the selector $LFSR_0$ is retained in the new Geffe structure, and the two source LFSRs are replaced by single $LFSR_3^+$, which embeds $LFSR_1$. The modified structure will create a sequence of selector bits such that the sequence has a cycle length T_2, where

$$T_2 = \mathrm{LCM}((2^{d0} -1), X)$$

where X is the period of $\mathrm{LFSR}_3{}^+$. Since $\mathrm{LFSR}_3{}^+$ is no longer a linear structure, its cycle length X is hard to compute by simple multiplications or additions. X is a value that highly depends on how compatible $\mathrm{LFSR}_3{}^+$ and its embedded LFSR are, when they work under the control of selector LFSR_0. For well constructed generators, X is able to reach the level of

$$X = (2^{d1} -1) \times (2^{d3} -1)$$

Where d_3 is the degree of the polynomial used to create $\mathrm{LFSR}_3{}^+$. Thus, the upper bound of the cycle length T_2 of the new Geffe generator is calculated to be

$$T_2 = (2^{d0} -1) \times (2^{d1} -1) \times (2^{d3} -1)$$

By the study of primitive polynomial embedding, $\mathrm{LFSR}_3{}^+$ very often can be built by a polynomial that its degree is larger than the degree of LFSR_1 and LFSR_2 in the original Geffe machine. Therefore, when the upper bound of cycle length T of the new Geffe generator can be compared with T, we have

$$T_2 \geq T$$

Thus, test sets produced by the new Geffe generators are very likely to produce a satisfactory fault coverage in the simulation at low cost. We are able to conclude that Modification 2 can be a promising solution to the problem of reducing hardware overhead and ensuring acceptable fault coverage.

3.5.3 Analysis of Modification 3

Flip Flop Reduction

Modification 3 lowers the hardware overhead by composing all three LFSRs, namely $LFSR_0$, $LFSR_1$ and $LFSR_2$ in the conventional Geffe machine. This modification transforms the original Geffe machine into a new machine $G_3(d_4[d_1])$. The number of flip-flops reduced by the modification can be calculated as follows:

Number of flip-flops in $G_3(d_4[d_1]) = d_4$

Number of flip-flop reduction = $(d_0 + d_1 + d_2) - d_4$

The percentage of reduction = $1 - \frac{d_4}{d_0 + d_1 + d_2} \times 100\%$

According to the calculation above, the amount of flip-flop reduction depends on the degree of characteristic polynomial used to build $LFSR_4^+$. By Modification 3, $LFSR_4^+$ can be implemented by a characteristic polynomial with larger degree such that $LFSR_1$ can embed in $LFSR_4^+$. Moreover, its degree d_4 must be a co-prime number to the degree of the characteristic polynomial of $LFSR_0$ and $LFSR_1$. As long as the d_4 is less than the sum of d_0, d_1 and d_2, reductions on hardware required to build a Geffe generator can be achieved. Table 3.6 shows four examples of this transformation. The results of flip flop reduction are also listed in the table.

Table 3.6 Characteristic polynomials used by Modification 3

# inputs in CUT n	$G(d_0, d_1, d_2)$			$G_3(d_4[d_1])$	# FF reduced
	$LFSR_0$	$LFSR_1$	$LFSR_2$	$LFSR_4^+$	
3	x^5+x^2+1	x^3+x^2+1	x^4+x+1	$X^{10}+x^5+ x^3+x^2+1$	3
4	x^5+x^2+1	x^4+x+1	x^5+x^2+1	$x^9+x^5+ x^4+x+1$	5
7	x^5+x^2+1	x^7+x^6+1	$x^8+x^4+ x^3+x^2+1$	$x^{13}+x^{12}+x^7+x^6+ 1$	7
9	x^5+x^2+1	x^9+x^4+1	$x^{10}+x^3+1$	$x^{16}+x^{14}+x^9+x^4+ 1$	8

[Assume x^5+x^2+1 is the characteristic polynomial of $LFSR_0$ for all Geffe machines.]

Prediction on Fault detection Performance

According to Modification 3, $d_4 \geq d_0 + d_2$. As a result, test patterns produced by the new machine $G_3(d_4[d_1])$ will have a cycle length T_3 such that its upper bound T_3_upper is

$$T_3\_upper = (2^{d1} -1) \times (2^{d4} -1) \geq (2^{d0} -1) \times (2^{d1} -1) \times (2^{d2} -1) = T$$

If there exists a Geffe generator that can be reconfigured according to the instructions given by Modification 3 and its output sequences have a cycle period which is close to T_3_upper, we can expect this new machine will have a similar performance as the original machine in the testing applications. Therefore, Modification 3 is a promising candidate for achieving cost-efficient chip-level testing.

3.6 Summary

The traditional Geffe generator can achieve the desired fault coverage in testing applications, but it is not practical to use for BIST since its implementation is complex and often involves a heavy hardware overhead. This tradeoff leads to a need for developing techniques to achieve good performance with a reasonable implementation cost.

All of the three modification techniques suggested in this chapter can successfully reduce the number of flip flops used by the LFSRs in a Geffe generator. The modified Geffe generators adopt the output scheme suggested by the existing model in [22]. They are able to produce a test vector instead of a single test bit per clock cycle for the fault simulation.

However, not all the modification techniques are candidates to ensure a desirable rate of fault detection once the generator is modified and reassembled in the simulation. Modification 1 directly eliminates the selector $LFSR_0$ in the original machine. However, this simple approach may sacrifice the good fault coverage that the original machine used to achieve in a circuit's fault simulation. Although in modification 2 and 3, a series of transformation steps must be followed in order to refine a Geffe generator, the improved structure has a high potential to preserve the performance of Geffe TPG.

Chapter 4

The Experiments

In Chapter 3, we introduce three different modifications to a Geffe generator; therefore its hardware overhead can be cut down to achieve the economics of the BIST. Our previous theoretical analysis on these modified Geffe generators suggests that only Modification 2 and Modification 3 provide promising solutions to the overhead problem introduced by original Geffe generators. The experiments discussed in this chapter can further our study of the behavior of these modified Geffe generators in testing applications. The experiments are also essential strategies to evaluate these Geffe modifications. They deliver concrete evidence that shows the impact of structural modifications on the conventional Geffe generators.

The experiments include two test methods: fault simulation and the test of cycle length and transition condition of the output sequence produced by different kinds of generators. We decide to apply these test methods to four groups of test pattern generators: LFSRs, the conventional Geffe generators and the reconstructed Geffe generators by Modification 2 and Modification 3. In this chapter, we give an overview of the layout of the experimental procedure. The detail experimental specification and the resources involved are discussed separately in Section 4.2 and Section 4.3.

4.1 Experimental Specification

4.1.1 Scope and Objectives

We earlier predicted on theoretical grounds that the design of Geffe Modification 2 and Modification 3 can generate test sequences that achieve high fault coverage in practical test time with minimal hardware overhead and performance penalty. Our goal of introducing the experimental procedure is to evaluate these new Geffe designs on practical grounds. The experimental procedure contains two phases: fault simulation and the evaluation of cycle length and transition condition of the test sequences produced by different kinds of generators.

One popular way to analyze a test generation system involves using fault simulation to evaluate its proposed test *T*. In the fault simulation, *T* is passed to a fault simulator to obtain the fault coverage, the ratio between the number of faults it detects and the total number of faults injected to the model [1, page 131]. The fault coverage is an important attribute that reflects the quality of a test. Therefore, the experiment of applying fault simulation to various kinds of Geffe modifications is able to provide critical evidence of the effectiveness of these new generators in real fault detection applications. Since the existing Geffe TPGs have different properties from linear TPGs, they can lead to better stuck-at fault coverage for sequential circuit testing, we use sequential circuits rather than combinational circuits throughout the fault simulation.

The research discussed in [21] and [22] shows that the cycle length and transition condition of the test produced by a generator have significant impacts on its fault coverage. The second phase of our experimental procedure is implemented to compute three attributes of a test; they are the cycle length, the number of different test pairs, and

the position of the last unique pair appearing in the test sequence. Unlike the fault simulation, in this experiment, we study the characteristic of modified Geffe generators based on analyzing these attributes.

Both test methods will exercise four groups of test pattern generators: LFSRs, the conventional Geffe generators and the reconstructed Geffe generators by Modification 2 and Modification 3. Therefore, the modified generators can be compared against not only the original Geffe generators but also the standard LFSR. These comparisons help to determine whether the suggested Geffe modifications are effective strategies to tailor a nonlinear test pattern generator which is more suitable for BIST.

4.1.2 Fault Simulation

Fault simulation is an important technique in test evaluation. It is frequently being used to determine the effectiveness or quality of a set of test sequences [1, page 131]. In the first part of the experiment, LFSRs, Geffe machines and their modifications will each produce a test file. Each of these test files contains a sequence of test patterns, and they are applied to a fault simulator for a quality evaluation.

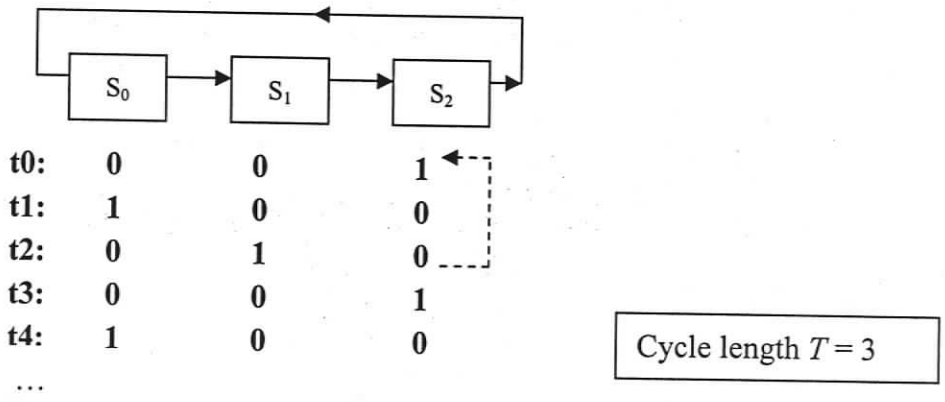
In general, the fault simulation involves three entities: the test produced by a TPG, the simulator, and a circuit model. In the fault simulation considered by our research, the tests T produced by different generators are used to target single stuck-at faults present in a given model, and the fault coverage associated with T is computed by the fault simulator. The fault simulator and the models under the simulation are discussed in detail later in the section describing the experimental design.

4.1.3 The Test of the Cycle Length and Transitions of Test patterns

In addition to the fault simulation, the test of the cycle length and transitions of test patterns counts the cycle period, the number of pattern transitions and the index position of the last transition in a sequence of patterns. Definitions associated with this experiment are given as follows.

- *Test pattern cycle length T, or cycle length in short.* A test pattern generator implemented by a finite state machine produces a sequence of binary patterns. If such a machine is initialized in a non zero state, it cycles through a sequence of states and eventually comes back to the initial state. Such a property is called self repeating. This also leads to the concept of cycle length which denotes the period that the sequence takes to repeat itself [3, page 62]. In other words, it is the number of states that a finite state machine can go through before it starts repeating the states [1, page 432]. For example, Figure 4.1 shows the cycle length of the LFSR described by the characteristic polynomial x^3+1.

Figure 4.1: The cycle length of a 3-bit LFSR



- *Test vector and the transition.* Test vector is another name for an individual pattern in the test sequence produced by a generator. A n-bit long test vector is denoted as $s = (s_1, s_2, \ldots, s_n)$, $s_p \in \{0, 1\}$, $1 \leq p \leq n$, and a transition corresponding to s is defined as

$$< (s_1, s_2, \ldots, s_n), (s_1^+, s_2^+, \ldots, s_n^+) >$$

where s_p^+ is the next state of s_p at each clock cycle.

If a test sequence contains k test vectors, then there are k-1 such transitions. The number of pattern transitions calculated in this experiment is the total number of different transitions contained in the test sequence. And the position of the last transition is the index location of the last different transition in the test set.

4.2 Design of the Experiment

4.2.1 Parameters of Test Pattern Generators

The previously introduced two tests are applied to pattern sequences generated by four groups of test pattern generators: LFSRs, the conventional Geffe generators and the reconstructed Geffe generators by Modification 2 and Modification 3. A table of all characteristic polynomials used by these generators is given in Appendix 1.

LFSR. Type 2 LFSR is used to produce test patterns for the experiment. An n-degree LFSR has n cells and is constructed from a characteristic polynomial of degree n. If this characteristic polynomial is primitive, the corresponding LFSR can cycle through all

possible non-zero states, therefore its cycle length is 2^n-1 [1, page 432]. Because such LFSR can reach the maximum cycle length, they are often chosen as the TPG in many testing scenarios. In this experiment, we only consider LFSRs which are built from primitive characteristic polynomials with the smallest number of terms (i.e. minimal weight). These characteristic polynomials are listed in [3].

In order to use a minimum amount of hardware to implement the TPG, the m-degree LFSR is chosen to simulate an m-input circuit. All LFSRs in the experiment are initialized in a non zero state. In the initialization, the first n-1 cells of a LFSR are set to logic '0', and the last cell has logic '1' instead.

The Geffe generator. The Geffe generator for this experiment is constructed using the structure demonstrated in section 3.4. The characteristic polynomials for all three LFSRs in a Geffe generator are primitive. If the three LFSRS have a degree which is relatively prime to the degree of the other two LFSRs, the Geffe generator constructed will produce maximum length sequences. However, in order to design a generator with such properties, a considerable amount of pre-calculation on the degree of the three LFSRs is required, and very often, the answer is not unique. Therefore, we choose a slightly simpler way to avoid the complicated degree calculation. For testing an n-input circuit, both $LFSR_1$ and $LFSR_2$ must have degree at least n. There is no such restriction for the selector $LFSR_0$. In our experiment, we use the n-degree LFSR for $LFSR_1$, and $(n+1)$-degree LFSR for $LFSR_2$. For $LFSR_0$, we choose x^5+x^2+1 as its characteristic polynomial. Consider the following an example. For an 3-input circuit, the three LFSRs are

$LFSR_0$: LFSR with the characteristic polynomial of x^5+x^2+1;

$LFSR_1$: LFSR with the characteristic polynomial of x^3+x+1;

$LFSR_2$: LFSR with the characteristic polynomial of x^4+x+1.

If the choice of the polynomials for a Geffe generator is specified as above, the cycle period of this machine is calculated to be

$$T = LCM[(2^5 -1), (2^n -1), (2^{n+1} -1)].$$

LCM refers to Least Common Multiple.

Modified Geffe generator of version 2. A standard Geffe generator can be modified according to the Modification 2, and the experiment will examine the test patterns generated by such a new machine. Because the modified Geffe generator is the reconfiguration of the existing model, we would like to reuse the original components as much as we can. The redefined structure contains two sub-machines: $LFSR_0$ and $LFSR_3^+$. The $LFSR_0$ in the new Geffe machine reuses the $LFSR_0$ in the old one. However, $LFSR_3^+$ is new to the Geffe structure; new characteristic polynomial must be selected for the $LFSR_3^+$. We start with the $LFSR_1$ in the original generator, and find its embedding in some reasonable larger LFSR, and use this LFSR as $LFSR_3^+$. When multiple such LFSR can be found, we select one at random. Consider the polynomial selection for the new Geffe generator, $G_2(5, 7[3])$, modified from G(5, 3, 4).

The three LFSRs in G(5, 3, 4) are:

$LFSR_0$: LFSR with the characteristic polynomial of x^5+x^2+1;

$LFSR_1$: LFSR with the characteristic polynomial of x^3+x+1;

$LFSR_2$: LFSR with the characteristic polynomial of x^4+x+1;

After the conversion, the two LFSRs in $G_2(5, 7[3])$ are:

$LFSR_0$: LFSR with the characteristic polynomial of x^5+x^2+1;

$LFSR_3^+$: LFSR with the characteristic polynomial of $x^7+x^5+\underline{x^3+x+1}$;

[Note: the part of $LFSR_3^+$ with an underscore is the $LFSR_1$ embedding.]

Modified Geffe generator of version 3. A standard Geffe generator can be also modified by Modification 3. The new Geffe machine has only one synthesis LFSR called $LFSR_4^+$. It is constructed according to the procedure described in Section 3.5.3. Consider an example of the new Geffe generator, $G_3(10[3])$, which is modified from the same Geffe machine as last example. The $LFSR_4^+$ in $G_3(10[3])$ is:

$LFSR_4^+$: LFSR with the characteristic polynomial of $x^{10}+x^4+\underline{x^3+x+1}$.

[Note: the part of $LFSR_4^+$ with an underscore is the $LFSR_1$ embedding.]

4.2.2 Benchmark Set

The ISCAS'89 benchmark set is used in this research to compare the performance among different types of TPGs. ISCAS'89 benchmark set differs than the ISCAS'85 benchmark set because it only consists of sequential digital circuits [6]. It was created for the ISCAS 89 Special Session on Sequential Test Generation [4]. It is a set of 31 sequential digital circuits [6]. Table 4.1 provides a summary of the internal details of ISCAS 89 benchmark circuits. The table includes the number of inputs and outputs, number of memory elements (D Type flip flops), number of gates and collapsed stuck-at faults, and the function of each circuit. The circuits described in the table are sorted according to the number of primary inputs.

Table 4.1: ISCAS' 89 benchmark circuits

Circuit name	Primary input	Primary output	D-type flip flop	# of gates	# of faults	Circuit function
s298	3	6	14	119	308	Traffic light controller, based on PLD
s382	3	6	21	158	399	Removed redundancies from s400
S400	3	6	21	162	424	Traffic light controller
s444	3	6	21	181	474	Traffic light controller
s526n	3	6	21	194	553	Removed redundancies from s526
s526	3	6	21	193	555	Traffic light controller
s27	4	1	3	10	32	Unclear
s386	7	7	6	159	384	Controller
s1488	8	19	6	653	1486	Removed redundancies from s1238
s1494	8	19	6	647	1506	Controller
s344	9	11	15	160	342	Removed redundancies from s349
s349	9	11	15	161	350	4-bit multiplier
s208	11	2	8	96	215	Digital fractional multiplier, based on PLD
s38584	12	278	1452	19253	36305	Real-chip based and rely on partial scan
s1196	14	14	18	529	1242	Removed redundancies from s1238
s1238	14	14	18	508	1355	A combinational circuit with randomly inserted FFs
s15850	14	87	597	9722	11727	Real-chip based and rely on partial scan
s953	16	23	29	395	1079	Controller
S1423	17	5	74	657	1515	Unclear
s820	18	19	5	289	850	Removed redundancies from s832
s832	18	19	5	287	870	Based on PLD
s420	19	2	16	196	430	Digital fractional multiplier
s510	19	7	6	211	564	Controller
s9234	19	22	228	5597	6927	Real-chip based on rely on partial scan

Circuit name	Primary input	Primary output	D-type flip flop	# of gates	# of faults	Circuit function
s38417	28	106	1636	22179	31180	Real-chip based on rely on partial scan
s13207	31	121	669	7951	9815	Real-chip based on rely on partial scan
s641	35	24	19	379	467	Removed redundancies from s713, based on PLD
s713	35	23	19	393	581	Based on PLD
s838	35	2	32	390	857	Digital fractional multiplier
s5378	35	49	179	2779	4603	Unclear
s35932	35	320	1728	16065	39094	Unclear

All ISCAS'89 benchmark circuits are described in a new netlist format called BENCH [5]. Figure 4.2 shows an example of the netlist format of circuit s27.

Figure 4.2: The netlist of s27

```
# 4 inputs
# 1 outputs
# 3 D-type flipflops
# 2 inverters
# 8 gates (1 ANDs + 1 NANDs + 2 ORs + 4 NORs)

INPUT(G0)
INPUT(G1)
INPUT(G2)
INPUT(G3)

OUTPUT(G17)

G5 = DFF(G10)
G6 = DFF(G11)
G7 = DFF(G13)

G14 = NOT(G0)
G17 = NOT(G11)

G8 = AND(G14, G6)

G15 = OR(G12, G8)
G16 = OR(G3, G8)

G9 = NAND(G16, G15)

G10 = NOR(G14, G11)
G11 = NOR(G5, G9)
G12 = NOR(G1, G7)
G13 = NOR(G2, G12)
```

4.2.3 HOPE Fault Simulator

For simulating single stuck-at faults in sequential circuits, HOPE is the simulator used in this research. HOPE is a later generation of PROOFS, a fast and memory efficient fault simulator [15]. Both of the two sequential circuit fault simulators were developed by Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute & State University [11, 15]. Different from PROOFS, HOPE employs the parallel version of the single fault propagation technique [11].

In this experiment, HOPE reads circuits in ISCAS'89 netlist format, and with test pattern files generated by different generators, it computes the corresponding fault coverage for the simulation.

4.2.4 Additional Programs for Test Sequence Evaluation

Two additional programs, *period.pl* and *pairs.pl*, are written for analyzing the sequences generated by different types of TPGs. Since these test sequences are stored in files of simple text format, the two programs are implemented in Perl language for a fast computation. Program *period.pl* is a function to determine the period or cycle length of a test sequence, while program *pairs.pl* is responsible for computing the number of transitions in the test sequence and the location of the last transition. The following are the algorithms for the two programs.

period.pl

```
input: pattern_array, an array of test vectors
output: period, Integer

variables:
first_pattern: String
cur_pattern: String
all_patterns: String
num_matches: Integer

//Convert pattern_array into a string of patterns. Patterns are separated by a blank.
all_patterns = join(" ", pattern_array);

//compute period
period=1;
first_pattern=pattern_array[0];
cur_pattern=first_pattern;
for i=1, i<pattern_array.length do
        if pattern_array[i] != first_pattern then
                period ++;
                cur_pattern=cur_pattern concatenate " " concatenate pattern_array[i];
        else
                 num_matches=( splite(/cur_pattern/, all_patterns ) ).length
                if pattern_array.length <= num_matches*period then
                        return period;
                else
                        period ++;
                        cur_pattern=cur_pattern concatenate " " concatenate pattern_array[i];
return period;
```

pairs.pl

```
input: pattern_array, an array of test vectors
output: num_trans, Integer
        last_tran_index, Integer

variables:
all_trans_str: String
cur_tran: String

last_tran_index=1;
num_trans=1;
all_trans_str= pattern_array[0] concatenate " " concatenate pattern_pattern[1] concatenate "|";

for i=0, i<pattern_array.length do
        cur_tran= pattern_array[i] concatenate " " concatenate pattern_pattern[i+1] concatenate "|";
        if all_trans_str has no match of cur_tran then
                num_trans++;
                last_tran_index=i++;
```

```
                    all_trans_str=all_trans_str concatenate cur_tran concatenate "|";
return num_trans, last_tran_index;
```

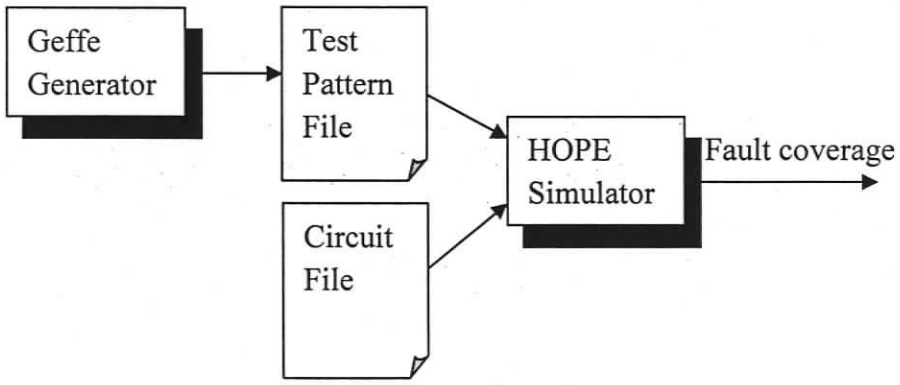
4.2.5 Experimental Setup

Fault simulation

In order to compute the fault coverage, a test pattern file and a circuit file must be available to the HOPE simulator at the same time. In this experiment, the test pattern file is a text file which consists of a list of binary test vectors. There are 2^{16}-1=32768 test vectors included in each test file used by the simulation. The general process of the fault simulation is illustrated in Figure 4.3.

The HOPE simulator has three different ways to initialize the flip flops in the circuit. At the beginning of the simulation, all flip flops of a certain circuit can be set to "0", "1", or "undefined". In the first and second phase of the fault simulation, all flip-flops are set to "0" and "1" respectively. In the last phase of the simulation, all flip-flops have undefined value.

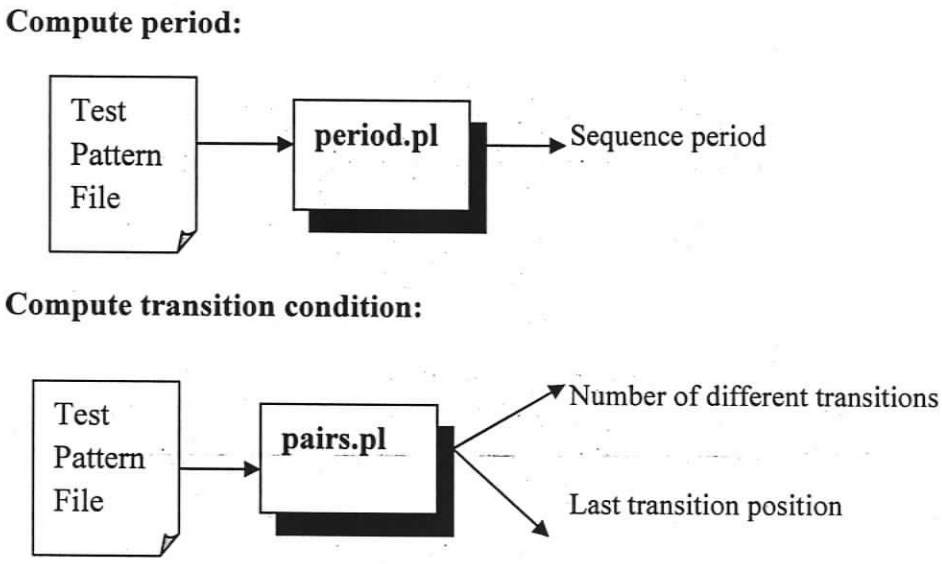
Figure 4.3: Fault simulation procedure



Test Sequence Evaluation

Every test pattern file used by the fault simulation is passed into period.pl and pairs.pl for further analysis on its cycle length and transition conditions. The general process of the test sequence evaluation is illustrated in Figure 4.4.

Figure 4.4: Test sequence evaluation procedure



4.3 Summary

Investigating the test pattern generation is often associated with the quality evaluation of the test sequence produced by the generator. The quality of a test sequence; however, can be described by important attributes such as fault coverage, cycle length and the

transition properties of the test sequence. The main goal of the experiment is to obtain these data, and to make the quality evaluation possible.

In this chapter, we introduce two test methods: the fault simulation and the test of the cycle length and transitions of a test sequence. These two experiments are designed to uncover the nature of modified Geffe generators. In the fault simulation, the pseudo-random test set produced by each of these generators is used to stimulate ISCAS'89 benchmarks, and, using HOPE fault simulator, we can evaluate the number of single stuck-at faults which are covered by this set. In addition to the fault simulation, we propose another test method to reveal the cycle length and the transition conditions of a test set. The results obtained from both experiments are important factors which describe the quality of a test set.

In the experiment, both test methods will exercise four groups of test pattern generators: LFSRs, the conventional Geffe generators and the reconstructed Geffe generators by Modification 2 and Modification 3. Therefore, the modified generators can be compared against not only the original Geffe generators but also the linear machines. All experimental results as well as the analysis of these results are discussed in the next chapter.

Chapter 5

Experimental Results and Analysis

In this chapter, we present the results of the two primary experiments: the fault simulation, and the test of the cycle length and transitions of test patterns. Using these results, we are able to compare the sequences produced by LFSRs, Geffe generators and two of our proposed modified Geffe generators. The Modification 2 and Modification 3 are two approaches designed to lower the hardware overhead of a traditional Geffe generator. The properties of each of the modified generators are then carefully studied and summarized in the following sections.

In Section 5.2, the redesigned Geffe generators are tested for stuck-at-fault simulation of the ISCAS'89 benchmark set. The simulation results are analyzed and conclusions are made in this section. In addition to the study of fault detection ability of the modified generators, Section 5.3 provides a conclusion of the evaluation on the cycle length and transitions of the test sequences used in the fault simulation. The detailed experimental results are given in Appendix 3.

5.1 Fault Simulation Results

In this section, we discus the fault simulation results of three categories of test pattern generators. They are the existing pattern generators (LFSRs and Geffe generators), modified Geffe generator of version 2, and modified Geffe generator of version 3. According to this classification, the layout of this section includes three parts, namely the simulation of existing generators, the simulation of modified Geffe of version 2, and the simulation of modified Geffe of version 3.

We start by presenting the experimental results of the two types of well-known generators: LFSRs and Geffe generators. The simulation results of LFSRs provide us with the lower bound of the fault coverage, which as they are nonlinear machines, the modified Geffe generators should exceed. On the other hand, we treat the simulation results of the original Geffe generators as the target bound, which the modified versions of Geffe generators try to achieve. Most importantly, this section also presents the results of the two Geffe modifications and their comparison to the two well-known generators. Our analysis of the experimental results focus on two aspects: the maximum fault coverage and how quickly a generator can reach the maximum fault coverage.

At the beginning of the analysis, our notation is stated as it is frequently referred to by the tables and charts included in this section.

1) ***n*:** it is $\log_2$(number of test patterns). It is used for a clear display of experimental results when plotting these results onto a chart.
2) **2^n patterns**: it denotes the number of test patterns used in the fault simulation.
3) **L(d_1):** it denotes a LFSR. d_1 is the degree of the LFSR, and d_1 is equal to the number inputs of the circuit in the simulation.

4) **G(d_0, d_1, d_2):** it denotes the original Geffe generator (Chapter 2, Figure 2.6) used in this section, where

   d_0 = the degree of $LFSR_0$,

   d_1 = the degree of $LFSR_1$, and

   d_2 = the degree of $LFSR_2$.

5) **G_2(d_0, d_3[d_1]):** it denotes the modified machine of G(d_0, d_1, d_2) based on Modification 2, where d_3 is the degree of $LFSR_3^+$.

6) **G_3(d_4[d_1]):** it denotes the modified machine of G(d_0, d_1, d_2) based on Modification 3, where d_4 is the degree of $LFSR_4^+$.

7) **FF->*i* (*i*=0, 1, or UD)**: it denotes the flip-flops of the circuit are initially set to logic 0, 1 or undetermined (UD) for the simulation.

8) **AVG**: it denotes the average of fault coverage from the simulations with the three different types of flip-flop initializations.

9) **Fault coverage limit**: it is the maximum fault coverage which can be achieved by a generator in the fault simulation.

10) **Num. of flip-flops**: it stands for the number of flip-flops used in a test pattern generator.

5.1.1 LFSRs and Standard Geffe Generators

The fault simulation of sequential circuits with test patterns generated by LFSRs and standard Geffe generators is delivered by the HOPE simulator. We use test sequences produced by LFSRs and Geffe machines to simulate the ISCAS' 89 benchmark circuits, whose number of inputs is less than 20. In total, there are 24 of these circuits, and the number of inputs for these circuits has a range of 3 to 19. In order to simulate all of these

circuits, we obtain 12 suitable test sequences from LFSRs and 12 suitable test sequences from conventional Geffe generators. Each test sequence is fed into the simulator the circuits to simulate three times with the circuit's flip-flops are initially set to logic 0, 1 or unknown.

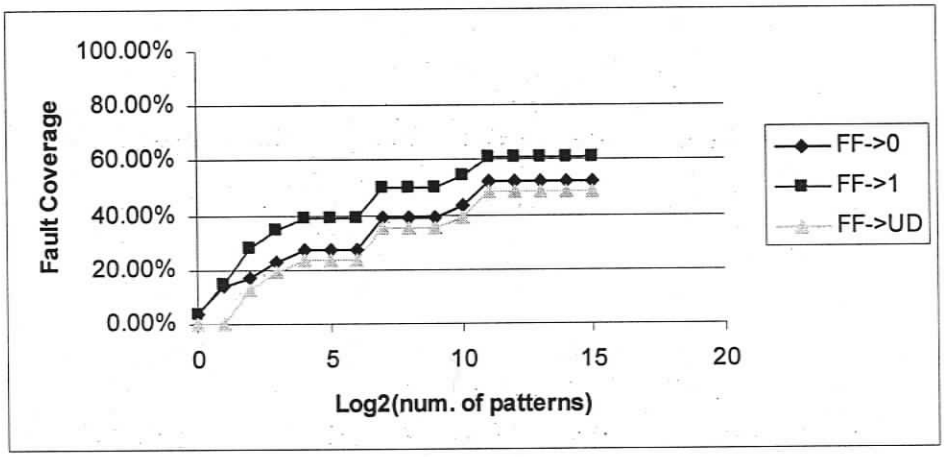
As a typical example, the stuck-at-fault coverage of LFSRs and Geffe generator are shown in Table 5.1. Table 5.1 lists all the fault coverage of L(3) and G(5,3,4) on circuit s298 at the points of 2^n ($0 \le n \le 15$) test patterns respectively. In this table, L(3) reaches its maximum fault coverage (24.03%) at 2^4=16 test patterns, while G(5,3,4) reaches its maximum fault coverage (52.27%) at 2^{11}=2048 test patterns, when flip-flop of s298 are pre-set to logic 0.

Table 5.1 Fault coverage for s298

n	2^n patterns	L(3)				G(5, 3, 4)			
		FF->0	FF->1	FF->UD	AVG	FF->0	FF->1	FF->UD	AVG
0	1	3.90%	3.90%	0.00%	**2.60%**	3.90%	3.90%	0.00%	**2.60%**
1	2	13.64%	14.61%	0.00%	**9.42%**	13.64%	14.61%	0.00%	**9.42%**
2	4	17.86%	28.25%	12.66%	**19.59%**	16.88%	28.25%	12.66%	**19.26%**
3	8	19.48%	31.82%	16.56%	**22.62%**	22.73%	34.74%	19.48%	**25.65%**
4	16	24.03%	35.71%	20.78%	**26.84%**	26.95%	38.96%	23.70%	**29.87%**
5	32	24.03%	35.71%	20.78%	**26.84%**	26.95%	38.96%	23.70%	**29.87%**
6	64	24.03%	35.71%	20.78%	**26.84%**	26.95%	38.96%	23.70%	**29.87%**
7	128	24.03%	35.71%	20.78%	**26.84%**	38.96%	49.68%	35.07%	**41.23%**
8	256	24.03%	35.71%	20.78%	**26.84%**	38.96%	49.68%	35.07%	**41.23%**
9	512	24.03%	35.71%	20.78%	**26.84%**	38.96%	50.33%	35.07%	**41.45%**
10	1024	24.03%	35.71%	20.78%	**26.84%**	43.51%	54.55%	39.29%	**45.78%**
11	2048	24.03%	35.71%	20.78%	**26.84%**	52.27%	61.36%	48.70%	**54.11%**
12	4096	24.03%	35.71%	20.78%	**26.84%**	52.27%	61.36%	48.70%	**54.11%**
13	8192	24.03%	35.71%	20.78%	**26.84%**	52.27%	61.36%	48.70%	**54.11%**
14	16384	24.03%	35.71%	20.78%	**26.84%**	52.27%	61.36%	48.70%	**54.11%**
15	32768	24.03%	35.71%	20.78%	**26.84%**	52.27%	61.36%	48.70%	**54.11%**

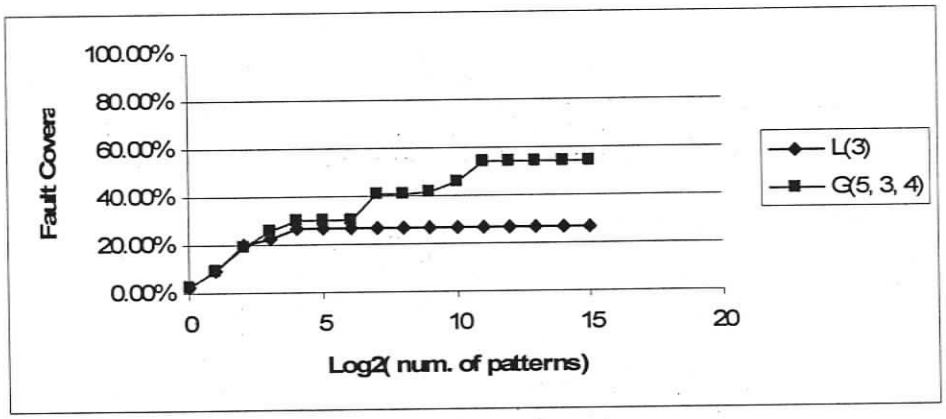
Figure 5.1 is a plot of the simulation results of G(5,3,4). This plot shows three curves from three different simulations with specific flip-flop setting. The curves on the plot represent the growth of fault coverage as the number of test patterns increases. For instance, G(5,3,4) achieves the best fault coverage when the circuit's flip-flops are initially set to 1; while the worst result is obtained when the flip-flop's initial setting is unknown.

Figure 5.1 Fault coverage of G(5,3,4) on circuit s298



For a general comparison of the fault detection performance of L(3) and G(5,3,4), we can average their fault coverage collected from three different flip-flop initializations. The results are shown in Figure 5.2. Figure 5.2 clearly suggests that G(5,3,4) can produce better fault coverage than L(3) does.

Figure 5.2 Fault coverage comparison of L(3) and G(5,3,4) on circuit s298



Observing all of the 24 circuits' simulation results, we discover several properties of the fault coverage achieved by LFSRs and Geffe generators. The stuck-at-fault coverage of both types of generators grows gradually as more and more test patterns are used in the simulation. However, their rates of growth progress quite differently. Compared with LFSRs, conventional Geffe generators often achieve a given level of fault coverage with a smaller number of patterns, and result in higher fault coverage when the number of test patterns stops at 2^{15} patterns. This property also accords with the research findings given in [22].

5.1.2 Modified Geffe Generator of Version 2

Conventional Geffe generators are modified according to the approach discussed in Section 3.5.3. The test sequences generated by modified Geffe generators of version 2 are exercised under the same simulation environment as the above. Their fault coverage

is collected in order to study the new machine's ability of detecting faults in the scenario of testing sequential circuits.

We consider the simulation result of $G_2(5, 7[3])$, the modification of G(5,3,4), as an example. Its fault coverage on circuit s298 is shown in Table 5.2. For all three types of flip-flops settings, $G_2(5, 7[3])$ reaches its fault coverage limit at 2^{12} test patterns. However, our other simulation cases show that not all modified generators reach their maximum fault coverage at a same point when flip-flops of the circuit are initialized differently for the simulation.

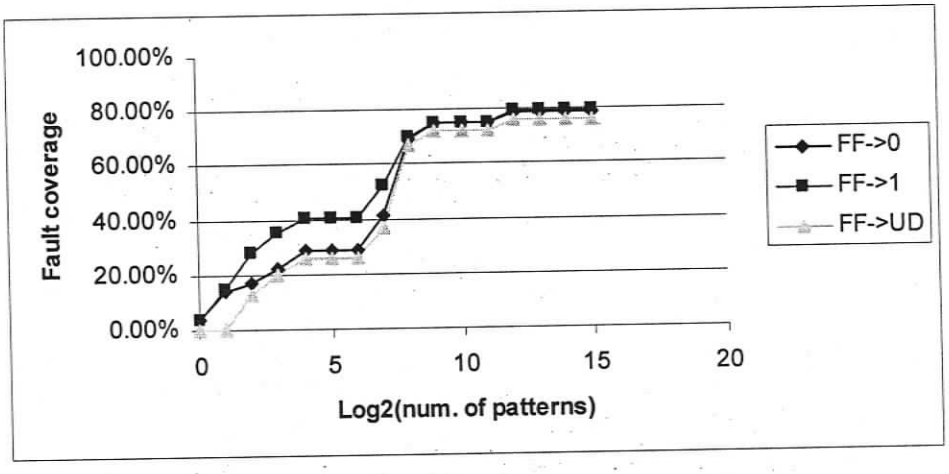
Table 5.2 Fault coverage of $G_2(5, 7[3])$ on s298

n	2^n patterns	$G_2(5, 7[3])$			
		FF->0	FF->1	FF->UD	AVG
0	1	3.90%	3.90%	0.00%	**2.60%**
1	2	13.64%	14.61%	0.00%	**9.42%**
2	4	16.88%	28.25%	12.66%	**19.26%**
3	8	21.75%	35.07%	19.81%	**25.54%**
4	16	28.90%	40.26%	25.65%	**31.60%**
5	32	28.90%	40.26%	25.65%	**31.60%**
6	64	28.90%	40.26%	25.65%	**31.60%**
7	128	41.23%	52.27%	37.01%	**43.51%**
8	256	69.48%	70.13%	66.56%	**68.72%**
9	512	74.68%	75.00%	71.75%	**73.81%**
10	1024	75.00%	75.33%	72.08%	**74.13%**
11	2048	75.00%	75.33%	72.08%	**74.13%**
12	4096	78.90%	79.22%	75.97%	**78.03%**
13	8192	78.90%	79.22%	75.97%	**78.03%**
14	16384	78.90%	79.22%	75.97%	**78.03%**
15	32768	78.90%	79.22%	75.97%	**78.03%**

Fault coverage limit point

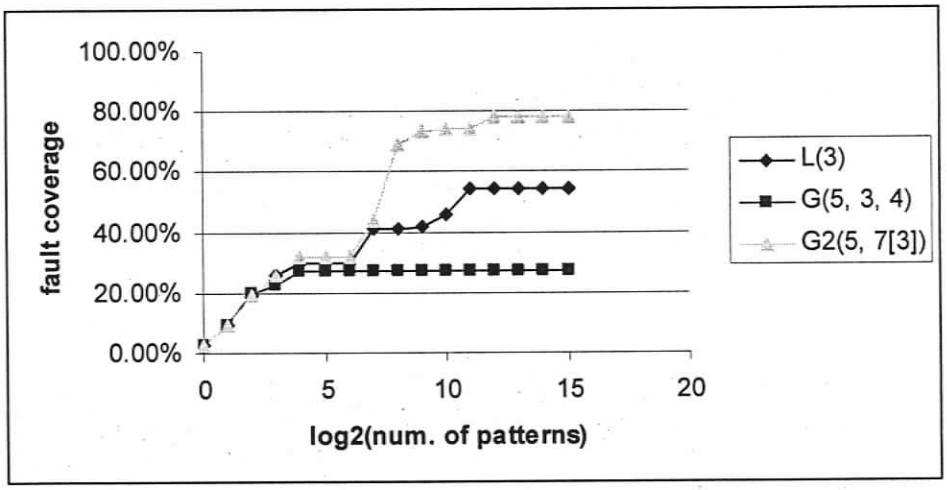
In Figure 5.3, we demonstrate a common character that L(3), G(5,3,4), and $G_2(5, 7[3])$ have regarding to the relationship of fault coverage and flip-flop initialization. When the flip-flops are initially set to the unknown state, all three machines obtain the lowest fault coverage compared with the results from the other two flip-flop settings. And when the flip-flops are initially set to"1", all three machines obtain the highest fault coverage compared with the results from the other two flip-flop settings.

Figure 5.3 Fault coverage obtained by $G_2(5, 7[3])$ on circuit s298



For $G_2(5, 7[3])$, an average value can also be calculated by averaging the fault coverage in the third, 4th and 5th column of Table 5.2. Comparing this result with the average fault coverage of L(3) and its original machine G(5,3,4), we found that the average fault coverage of $G_2(5, 7[3])$ is the highest. Figure 5.4 shows such finding.

Figure 5.4 Average fault coverage of L(3), G(5,3,4) and G_2(5, 7[3]) on circuit s298



Does the above finding hold true for all simulation results? Besides s298, we studied 23 other ISCAS' 89 circuits (input less than 20). We consider the following analysis in order to obtain an idea of how well the new machine can behave in the testing scenario.

For each circuit, we group the fault coverage received at 2^{15} input test patterns produced by different types of generators and the average fault coverage is calculated for each group. These results can better reflect the general performance of different generators used in the fault detection. The average fault coverage for all simulated circuits is listed in Table 5.3. With a few exceptions, our results confirm that modified Geffe generators have much higher average fault coverage than LFSRs have (see Table 5.3).

We further analyze the difference between the average fault coverage collected before and after the structural modification for Geffe generators. The results from 13 out of 24 circuits agree on clear fault coverage increase achieved by our new machines. Although the remaining test results do not show such significant improvement, the fault coverage

from 8 other circuits show a reasonable degree of reduction (less than 6%). To demonstrate this point, Table 5.3 also includes a rating column. The given ratings are based on the average fault coverage difference between conventional Geffe generators and modified Geffe generator of version 2.

Table 5.3 Average fault coverage of 24 ISCAS' 89 circuits received at 2^{15} input patterns (3 types of TPGs)

circuit	LFSR		Conventional Geffe Generator		Modified Geffe Generator 2		
	Avg. FC	Num. of flip-flops	Avg. FC	Num. of flip-flops	Avg. FC	Rating	Num. of flip-flops
s298	26.84%	3	54.11%	12	78.03%	Good	12
s382	13.11%	3	14.78%	12	14.87%	Good	12
s400	12.66%	3	14.55%	12	14.55%	Good	12
s444	12.59%	3	13.43%	12	20.75%	Good	12
s526n	12.06%	3	12.24%	12	17.30%	Good	12
s526	12.01%	3	12.19%	12	17.24%	Good	12
s27	71.88%	4	100.00%	14	97.79%	Fair	12
s386	31.69%	7	69.27%	20	47.40%	Bad	14
s1488	48.48%	8	68.12%	22	56.37%	Bad	16
s1494	47.83%	8	67.29%	22	55.62%	Bad	16
s344	77.10%	9	97.76%	24	96.10%	Fair	16
s349	77.05%	9	97.24%	24	95.62%	Fair	16
s208	68.67%	11	69.43%	28	69.43%	Good	18
s38584	63.34%	12	62.64%	30	64.16%	Good	22
s1196	95.33%	14	98.39%	34	95.17%	Fair	22
s1238	90.11%	14	93.06%	34	92.26%	Fair	22
s15850	39.61%	14	37.23%	34	38.40%	Good	22
s953	68.71%	16	68.83%	38	68.83%	Good	24
s1423	38.26%	17	56.96%	40	53.16%	Fair	24
s820	40.28%	18	49.33%	42	46.90%	Fair	28
s832	39.35%	18	48.20%	42	45.94%	Fair	28
s420	57.37%	19	53.04%	44	55.09%	Good	26
s510	66.67%	19	66.67%	44	66.67%	Good	26
s9234	12.83%	19	15.42%	44	15.55%	Good	26

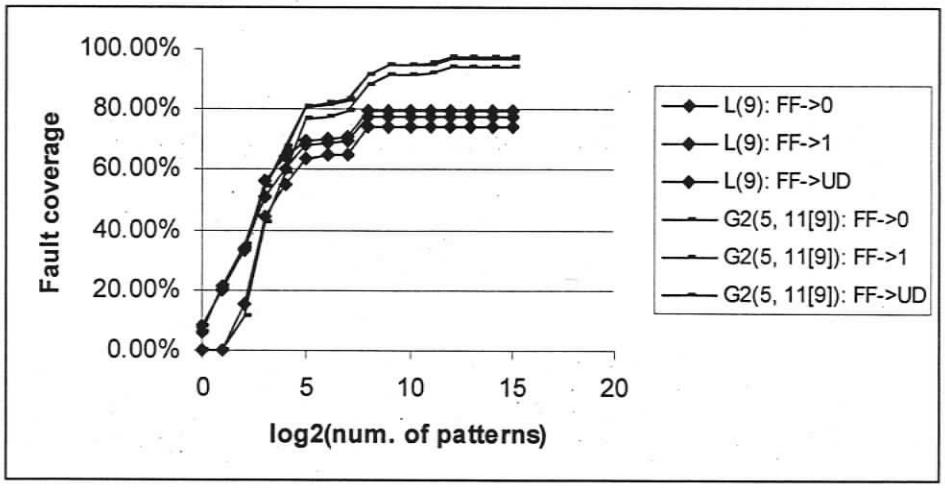
Note: Good – Avg. FC(modified Geffe generator 2) - Avg. FC(original Geffe generator) ≥ 0
Fair – Avg. FC(original Geffe generator) - Avg. FC(modified Geffe generator 2) $\leq 6\%$
Bad – Avg. FC(original Geffe generator) - Avg. FC(modified Geffe generator 2) $> 6\%$

With our analysis so far, we notice that performing analysis primarily on the fault coverage received at 2^{15} input patterns is not able to provide a precise description on the nature of pattern generation by modified Geffe machines. Therefore, we take our analysis to the next step. The effectiveness of a pattern generator in a fault simulation can be disclosed by how fault coverage rises when the generator contributes more and more test patterns to the simulation.

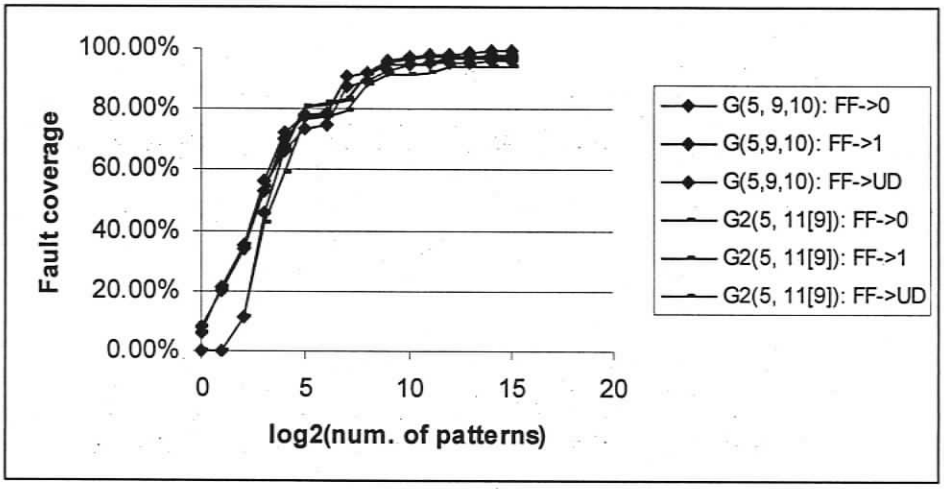
We use the fault simulation associated with circuit s344 as an example to review our analysis results (see Figure 5.5). In this fault simulation, we respectively use L(9), G(5,9,10) and G_2(5, 11[9]) as test pattern generators to simulate the existing faults of s344. For each generator, fault coverage at different numbers of input patterns is collected, and the results are exhibited in the two graphics given by Figure 5.5.

In graph (b) of Figure 5.5, the curves of G_2(5, 11[9]) and G(5,9,10) are very alike, but this does not occur when comparing the same attribute between G_2(5, 11[9]) and L(9) (see graph (a)). Graph (a) shows with all three flip-flop settings, both G_2(5, 11[9]) and L(9) start with a rapid increase in additional faults being detected by the test patterns they generate. However, as the number of test patterns increases, the fault coverage obtained by G_2(5, 11[9]) rises higher and separates from the fault coverage obtained by L(9). On the other hand, we observe a different situation from Graph (b). It shows in spite of the minor differences between the fault coverage for G_2(5, 11[9]) and G(5,9,10), that the two generators follow a similar pattern with respect to the fault coverage.

Figure 5.5 Fault coverage obtained by L(9), G(5,9,10) and G_2(5, 11[9]) on circuit s344



(a) L(9) vs. G_2(5, 11[9])



(b) G(5, 9, 10) vs. G_2(5, 11[9])

Most of the simulation results of ISCAS'89 circuits show similar characteristics to the new Geffe generators of version 2. The version 2 modification approach has a strong potential to preserve the character of the original Geffe machine, and this agrees on our fault coverage prediction in Section 3.6.2.

5.1.3 Modified Geffe Generator of Version 3

Another technique to modify Conventional Geffe generators is described in Section 3.5.3. It suggests that we can combine all three LFSRs of a particular Geffe generator into one slightly larger LFSR. We exam the test sequences generated by the generators modified according to this approach under the same simulation environment as the above. The fault coverage, then, is collected in order to study the new machine's ability of detecting fault in the scenario of testing sequential circuits. Since similar techniques are used by version 2 and version 3 modifications to refine the standard Geffe generator, our study of version 3 modification follows the same strategies as those used to evaluate the version 2 modification.

We start our analysis with a study in $G_3(7[3])$, a typical example of modified Geffe generators of version 3. Both generator $G_3(7[3])$ and $G_2(5, 7[3])$ are transformed Geffe generators but use different approaches for the transformation. Using $G_3(7[3])$ for the analysis is consistent with the analysis of $G_2(5, 7[3])$ given in Section 5.2.2; therefore, we can capture the similarity and difference between the two machines. Test patterns generated by $G_3(7[3])$ are input to circuit s298 for the simulation, the fault coverage received is shown in Table 5.4.

Table 5.4 Fault coverage obtained by $G_3(7[3])$ on s298

n	2^n patterns	$G_3(7[3])$ FF->0	FF->1	FF->UD	AVG
0	1	3.90%	3.90%	0.00%	**2.60%**
1	2	13.64%	14.61%	0.00%	**9.42%**
2	4	16.88%	28.25%	12.66%	**19.26%**
3	8	22.73%	36.04%	20.46%	**26.41%**
4	16	23.38%	36.04%	20.78%	**26.73%**
5	32	33.77%	45.13%	30.52%	**36.47%**
6	64	34.09%	45.46%	30.52%	**36.69%**
7	128	50.00%	59.74%	47.08%	**52.27%**
8	256	56.49%	65.26%	52.92%	**58.23%**
9	512	69.81%	70.46%	66.88%	**69.05%**
10	1024	69.81%	70.46%	66.88%	**69.05%**
11	2048	69.81%	70.46%	66.88%	**69.05%**
12	4096	69.81%	70.46%	66.88%	**69.05%**
13	8192	69.81%	70.46%	66.88%	**69.05%**
14	16384	69.81%	70.46%	66.88%	**69.05%**
15	32768	69.81%	70.46%	66.88%	**69.05%**

Fault coverage limit point

From Table 5.4, for all three types of flip-flops settings, $G_3(7[3])$ reaches the fault coverage limit at 2^9 test patterns. Compared with L(3), $G_3(7[3])$ meets its fault limit much later (for L(3), the limit is at 2^4). But, compared with G(5, 3, 4) and $G_2(5, 7[3])$, $G_3(7[3])$ meets its fault limit earlier (For G(5, 3, 4), the limit is at 2^{11}, and for $G_2(5, 7[3])$, the limit is at 2^{12}).

Although L(3), G(5,3,4), $G_2(5, 7[3])$, and $G_3(7[3])$ trigger the fault coverage limit at different points, the flip-flop initialization in the simulation affects the fault coverage associated with different generators in the same way. For instance, when the flip-flops are initially set to the unknown state, all four machines obtain the lowest fault coverage compared with the results from the other two flip-flop settings. And when the flip-flops

are initially set to"1", all four machines obtain the highest fault coverage compared with the results from the other two flip-flop settings

By averaging the different fault coverage caused by different flip-flop initializations, we are able to review the general performance of a generator in the fault simulation. The average fault coverage of c298 for L(3), G(5,3,4), G_2(5, 7[3]), and G_3(7[3]) generators are given in Table 5.5.

Table 5.5 Average fault coverage of 24 ISCAS' 89 circuits received at 2^{15} input patterns (4 types of TPGs)

Circuit	LFSR		Conventional Geffe Generator		Modified Geffe Generator 2			Modified Geffe Generator 3		
	Avg. FC	Num. of flip-flops	Avg. FC	Num. of flip-flops	Avg. FC	Rating	Num. of flip-flops	Avg. FC	Rating	Num. of flip-flops
s298	26.84%	3	54.11%	12	78.03%	Good	12	69.05%	Good	7
s382	13.11%	3	14.78%	12	14.87%	Good	12	14.87%	Good	7
s400	12.66%	3	14.55%	12	14.55%	Good	12	14.55%	Good	7
s444	12.59%	3	13.43%	12	20.75%	Good	12	18.43%	Good	7
s526n	12.06%	3	12.24%	12	17.30%	Good	12	15.91%	Good	7
s526	12.01%	3	12.19%	12	17.24%	Good	12	15.85%	Good	7
s27	71.88%	4	100.00%	14	97.79%	Fair	12	88.54%	Bad	9
s386	31.69%	7	69.27%	20	47.40%	Bad	14	47.40%	Bad	13
s1488	48.48%	8	68.12%	22	56.37%	Bad	16	55.03%	Bad	15
s1494	47.83%	8	67.29%	22	55.62%	Bad	16	54.23%	Bad	15
s344	77.10%	9	97.76%	24	96.10%	Fair	16	95.71%	Fair	16
s349	77.05%	9	97.24%	24	95.62%	Fair	16	95.24%	Fair	16
s208	68.67%	11	69.43%	28	69.43%	Good	18	69.43%	Good	17
s38584	63.34%	12	62.64%	30	64.16%	Good	22	63.06%	Good	19
s1196	95.33%	14	98.39%	34	95.17%	Fair	22	95.65%	Fair	23
s1238	90.11%	14	93.06%	34	92.26%	Fair	22	90.70%	Fair	23
s15850	39.61%	14	37.23%	34	38.40%	Good	22	37.00%	Fair	24
s953	68.71%	16	68.83%	38	68.83%	Good	24	65.06%	Fair	23
s1423	38.26%	17	56.96%	40	53.16%	Fair	24	44.51%	Bad	23
s820	40.28%	18	49.33%	42	46.90%	Fair	28	47.02%	Fair	25
s832	39.35%	18	48.20%	42	45.94%	Fair	28	46.05%	Fair	25
s420	57.37%	19	53.04%	44	55.09%	Good	26	57.73%	Good	25
s510	66.67%	19	66.67%	44	66.67%	Good	26	66.67%	Good	25
s9234	12.83%	19	15.42%	44	15.55%	Good	26	15.10%	Fair	25

Note: Good – Avg. FC(modified Geffe generator 3) - Avg. FC(original Geffe generator) ≥ 0

Fair – Avg. FC(original Geffe generator) - Avg. FC(modified Geffe generator 3) $\leq 6\%$

Bad – Avg. FC(original Geffe generator) - Avg. FC(modified Geffe generator 3) $> 6\%$

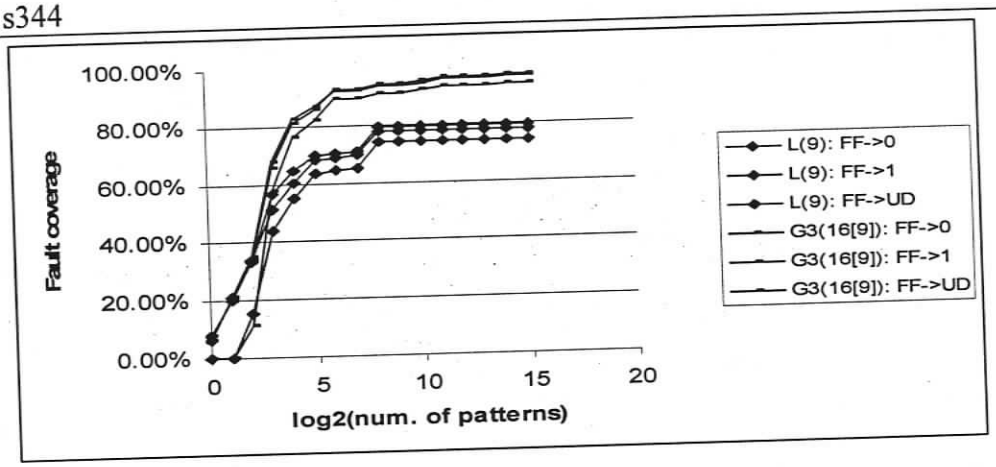
From Table 5.5, similar to version 2 modified Geffe generators, version 3 generators exhibit better performance than the LFSRs. However, compared the average fault coverage obtained by version 3 generators against the results of the original Geffe machines, the new generators do not lead to significant improvements in fault coverage. Although fault coverage of 19 out of 24 circuits is rank either "Good" or "Fair", we see a slight degradation of the performance for version 3 machines. This is quite understandable because a version 3 generator purely relies on single LFSR instead of three LFSRs as the original Geffe generators to perform the test pattern generation. When the hardware overhead is reduced massively, we expect a reasonable degree downgrade in its performance.

The average fault coverage of the 24 ISCAS' 89 circuits also shows a difference between version 2 Geffe generators and version 3 Geffe generators. Generally speaking, version 3 Geffe generators lead to slightly less impressive fault coverage than version 2 Geffe generators do. However, some simulated circuits such as s420 and s1196 are able to receive higher fault coverage when they are given test patterns produced by version 3 Geffe generators instead of version 2 Geffe generators.

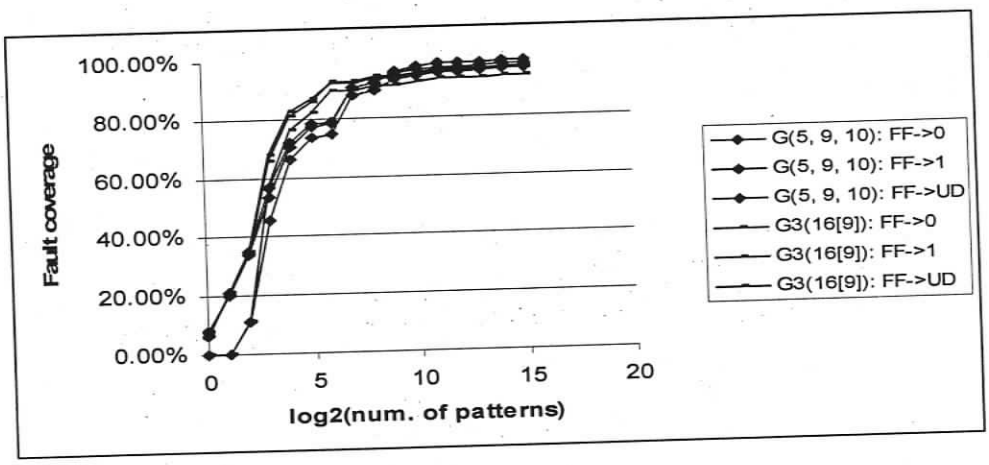
From the observation on the fault coverage changes when the number of input test patterns is increased, several other characteristics for version 3 Geffe generators are discovered. In the following, we use the fault simulation results of s344 to illustrate these characteristics. Similar to LFSRs, version 3 Geffe generators produce test patterns primarily based on one LFSR. Compared to the original Geffe generators and version 2 modified Geffe generators, version 3 modified Geffe generators are structurally closest to linear machines. However, the fault coverage associated with version 3 modified Geffe

generators increases differently from the fault coverage associated with LFSRs, as shown in Figure 5.6 (a). And, in fact, version 3 modified Geffe generators lead to a similar fault coverage increase as the original Geffe generators and version 2 modified Geffe generators. Figure 5.6 (b) and (c) shows this character.

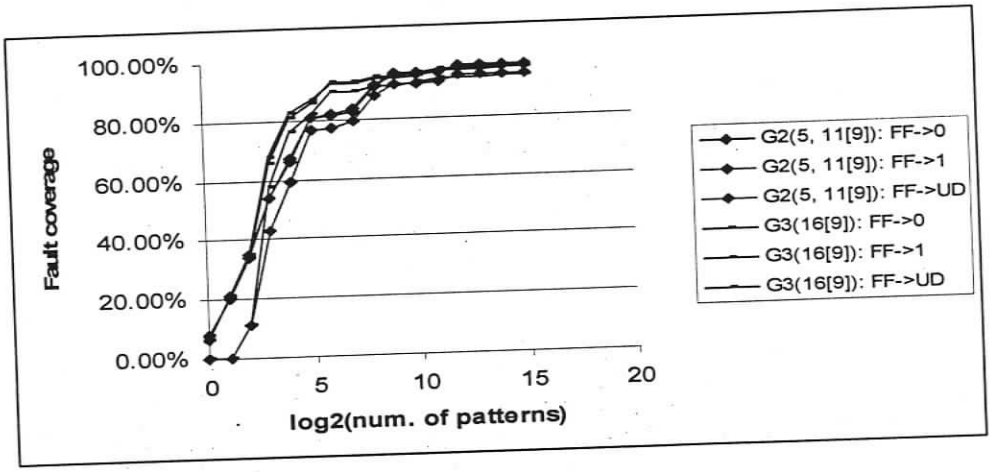
Figure 5.6 Fault coverage obtained by L(9), G(5,9,11), G_2(5, 11[9]) and G_3(16[9]) on circuit s344



(a) L(9) vs. G_3(16[9])



(b) G(5, 9, 10) vs. G_3(16[9])



(c) $G_2(5, 11[9])$ vs. $G_3(16[9])$

5.2 The Test of the Cycle Length and Transitions of Test Patterns

In this section, we evaluate the cycle length and transitions of test patterns produced by different types of test pattern generators. This may allow us to explain some of the fault simulation results discussed in Section 5.1.

5.2.1 The Cycle Length of Test Patterns

In the following, Table 5.6 lists the cycle length of all test patterns used in the fault simulation. Note that for a certain simulation in our experiment, the number of test patterns used is not greater than 2^{15}=32768. The cycle lengths in Table 5.6 are computed from the simulation test files, which store exactly 32768 patterns. The table contains some computation results of ">32768". This is because the sequences of 32768 test patterns are the maximum length test sequences used in our simulation, and the cycle length of these sequences is calculated thereby. Therefore, if the repetition search

reaches the 32768th test pattern with no repetition found, we mark its cycle length as ">32768" to indicate that the actual cycle length is greater than 32768.

Table 5.6 The cycle length of test patterns

Pattern length	LFSR		Conventional Geffe Generator		Version 2 Modified Geffe Generator		Version 3 Modified Geffe Generator	
	Notation	Period	Notation	Period	Notation	Period	Notation	Period
3	L(3)	7	G(5,3,4)	3255	$G_2(5,7[3])$	3937	$G_3(7[3])$	680
4	L(4)	15	G(5,4,5)	465	$G_2(5,7[4])$	3937	$G_3(9[4])$	121
7	L(7)	127	G(5,7,8)	>32768	$G_2(5,9[7])$	11811	$G_3(13[7])$	710
8	L(8)	255	G(5,8,9)	>32768	$G_2(5,11[8])$	19685	$G_3(15[8])$	24247
9	L(9)	511	G(5,9,10)	>32768	$G_2(5,11[9])$	31713	$G_3(16[9])$	>32768
11	L(11)	2047	G(5,11,12)	>32768	$G_2(5,13[11])$	15810	$G_3(17[11])$	21075
12	L(12)	4095	G(5,12,13)	>32768	$G_2(5,17[12])$	>32768	$G_3(19[12])$	>32768
14	L(14)	16383	G(5,14,15)	>32768	$G_2(5,17[14])$	19530	$G_3(23[14])$	>32768
16	L(16)	>32768	G(5,16,17)	>32768	$G_2(5,19[16])$	>32768	$G_3(23[16])$	>32768
17	L(17)	>32768	G(5,17,18)	>32768	$G_2(5,19[17])$	>32768	$G_3(23[17])$	>32768
18	L(18)	>32768	G(5,18,19)	>32768	$G_2(5,23[18])$	>32768	$G_3(25[18])$	>32768
19	L(19)	>32768	G(5,19,20)	>32768	$G_2(5,21[19])$	>32768	$G_3(25[19])$	>32768

For conventional Geffe generators, if the degrees of the three LFSRs in a generator are co-prime to each other, the larger these degrees are, the longer its pattern cycle length will be. However, from Table 5.6, we realize this is not always the case for modified Geffe generators. For example, $G_3(15[8])$ uses degree 15 LFSR to generate test patterns, and its patterns have a cycle length of 24247; while $G_3(17[11])$ use degree 17 LFSR to generate test patterns, but its pattern cycle length is 21075. This is because the MUX from the original Geffe generator is placed inside the LFSR to take advantages of using both the LFSR and its embedded LFSR. In doing so, the original pattern cycle for both LFSRs can be interrupted. If the feedback bits selected from the LFSR and its embedding are not able to make "good" transitions between the two pattern cycles, it is possible that

the cycle length for the modified machine becomes shorter. The pattern cycle length of the modified Geffe generator depends on how well the relevant LFSRs work together.

Table 5.6 shows the cycle length of the Geffe generator, the modified Geffe generator of version 2, and the modified Geffe generator of version 3 is greater than the cycle length of the LFSRs. This explains why LFSRs receive the lowest fault coverage amongst all the generators exercised in the simulation. The experimental results also suggest that both versions of modified Geffe generators have a shorter cycle length than the original Geffe generators. This discovery may give partial explanation to the situations that some of our modified generators are not able to deliver high performance in the fault simulation. Also, from Table 5.6, we can recognize that the cycle length of version 3 modified Geffe generators is not necessarily worse than the cycle length of version 2 modified Geffe generators (see the modifications of G(5, 8, 9): G_2(5,11[8]) and G_3(15[8]) . This provides indication that version 3 modified Geffe generators can perform as well as version 2 modified Geffe generators in the fault simulation.

Record Section 5.2.2. From the fault coverage comparison between version 2 Geffe generators and the conventional Geffe generators, we saw that version 2 Geffe generators possess the capacity for preserving the fault coverage which can be achieved by the original Geffe machines. However, according to Table 5.6, the cycle length of version 2 Geffe generators is shorter than the cycle length of the original machine. This result seems to provoke a contradiction to the conjecture about large cycle length and high fault coverage. Our argument to this is that, in the scenario of simulating sequential circuits, it is the variety of pattern transitions rather than the number of patterns that determines the performance of a sequence of patterns produced by a test pattern generator [22, page 66]. Therefore, in order to provide further evidence on why modified Geffe generators can

perform well, we have to look into the transition condition of test patterns generated by our new machines.

5.2.2 The Transitions of Test Patterns

Table 5.7 The transitions of test patterns

Pattern length	LFSR		Conventional Geffe Generator		Version 2 Modified Geffe Generator		Version 3 Modified Geffe Generator	
	Notation	Trans.	Notation	Trans.	Notation	Trans.	Notation	Trans.
3	L(3)	7	G(5,3,4)	63	$G_2(5,7[3])$	16	$G_3(7[3])$	16
		7		1125		114		37
4	L(4)	15	G(5,4,5)	207	$G_2(5,7[4])$	32	$G_3(9[4])$	30
		15		465		205		107
7	L(7)	127	G(5,7,8)	11456	$G_2(5,9[7])$	256	$G_3(13[7])$	220
		127		32767		2239		693
8	L(8)	255	G(5,8,9)	15788	$G_2(5,11[8])$	512	$G_3(15[8])$	512
		255		32768		3864		5760
9	L(9)	511	G(5,9,10)	17139	$G_2(5,11[9])$	1024	$G_3(16[9])$	1024
		511		32767		13815		9856
11	L(11)	2047	G(5,11,12)	21781	$G_2(5,13[11])$	3813	$G_3(17[11])$	3945
		2047		32767		15737		20947
12	L(12)	4095	G(5,12,13)	8550	$G_2(5,17[12])$	7609	$G_3(19[12])$	7683
		4095		32767		32749		32765
14	L(14)	16383	G(5,14,15)	31366	$G_2(5,17[14])$	13978	$G_3(23[14])$	19263
		16383		32737		19520		32767
16	L(16)	32767	G(5,16,17)	32507	$G_2(5,19[16])$	28217	$G_3(23[16])$	1099
		32767		32767		32767		1099
17	L(17)	32767	G(5,17,18)	32664	$G_2(5,19[17])$	30226	$G_3(23[17])$	4219
		32767		32767		32767		4219
18	L(18)	32767	G(5,18,19)	32704	$G_2(5,23[18])$	31581	$G_3(25[18])$	31604
		32767		32767		32767		32767
19	L(19)	32767	G(5,19,20)	32745	$G_2(5,21[19])$	32098	$G_3(25[19])$	32203
		32767		32767		32767		32767

Note: For each generator, there are two values in the Transition column (Trans.). The upper one is the number of different pairs of patterns found in 32768 test patterns and the bottom one is the position where the last new pair appears. For example, the 32768 test patterns produced by G(5,4,5) contains 207 different pairs of patterns and the last new pair appears in position 465 in the sequence.

Table 5.6 shows the transition evaluation results for all test patterns used in the fault simulation. These results indicate that the conventional Geffe generators and their modifications have richer pattern pairs than linear generators. This leads to good performances of the non-linear machines in fault simulation for sequential circuits. However, when we study the transitions of type 2 and type 3 modified Geffe generators, we find a few interesting surprises in our experimental results. For example, compared with the conventional Geffe generators, the modified generators do not show advantages in their transitions in general. But, there are situations where generators have a greater number of transitions; however, they don't lead to high fault coverage (see the fault simulation result for circuit s38584) either.

Our transition evaluation results are not able to give strong explanations on why modified Geffe generators such as version 2 generators can achieve a similar fault coverage as the original Geffe generators. We are seeking opportunities for future researches to explore the analysis on the Geffe generator modification.

5.3 Summary

In this chapter, we presented the experimental results of the fault simulation and the test of transition properties of test patterns used in the simulation. The performance of modified Geffe generators is evaluated based on the quality and the effectiveness of the test patterns they generate.

Fault coverage from the fault simulation on ISCAS' 89 circuits is studied at first. We discover that version 2 and version 3 modified Geffe generators win the LFSRs as the

modified generators lead to better fault coverage. In additional, the simulation results indicate the version 2 modified Geffe generators are capable to maintain the fault detection ability of the original Geffe generators. However, as the number of flip-flops in Geffe generators being further reduced, the fault coverage of version 3 modified Geffe generators receives a reasonable degree of degradation.

We also analyze the experimental results from the test of the cycle length and transitions of test patterns generated by various types of generators. From the cycle length and transition properties of the test patterns, explanation is given to answer why modified Geffe generators, rather than LFSRs, are superior generators for testing applications. More interestingly, some results from this part of experiments raise conflicts to the discoveries which we perceive from the fault simulation results. They suggest future researches.

Chapter 6

Conclusions

The goal of this research is to reduce the hardware overhead of the conventional Geffe generator without having serious effects on its original fault detection ability, so that it is a more suitable test pattern generator for the BIST. Several structural modification approaches have been designed to accomplish this goal. In the following, we summarize the main contributions of our research, and raise some of the possible directions for future work.

6.1 Contributions

Conventional Geffe generators are nonlinear FSMs, which incorporate a 2-to-1 multiplexer and three LFSRs (one selector LFSR, and two source LFSRs) in their structure. In testing applications such as delay fault and stuck-at fault detection, the conventional Geffe generators often lead to high fault coverage. However, such excellent performance is achieved with one tradeoff that a significant amount hardware resource is required by the generators. Additional circuitry introduced by using the conventional Geffe generator as TPG raises the complexity issue to the design of testing digital systems. This creates a challenge to the implementation of reliable BIST schemes.

The contributions of this thesis can be summarized as follows:

- Three modification methods: the removal of the selector LFSR (Modification 1), the modification on the two source LFSRs (Modification 2), and combining all three LFSRs (Modification 3), are developed by our research in order to solve the overhead problem associated with conventional Geffe generators. Our careful analysis of these three approaches indicates Modification 1 is the simplest and the most intuitive approach, yet it can cause serious fault coverage degradation. Compared with Modification 1, the other two methods are slightly more complicated solutions to the problem; however, they offer much better possibilities of delivering high fault coverage at low cost. Although the latter two modifications adopt the same polynomial embedding technique to refine the structure of the Geffe generator, Modification 3 achieves a larger percentage of flip-flop reduction than Modification 2.

- Both of Modification 2 and Modification 3 are applied to the conventional Geffe generators. The newly constructed machines are exercised in the fault simulation of twenty-four ISCAS' 89 benchmark circuits. Our simulation results show that the Geffe generator modified by version 2 behaves very similarly to the unmodified machine, and it is able to preserve the fault coverage achieved by the original. However, we observe slightly weaker performance from Modification 3 generators in the simulation. This situation is expected as Modification 3 results in a larger reduction in the number of flip-flops. In general, both versions of modified Geffe generators can lead to satisfied simulation results, which are difficult to achieve by LFSRs.

- The version 2 and version 3 modified Geffe generators produce test patterns, which are not only exercised in the fault simulation, but also studied for their cycle length and transition properties. The test patterns produced by the new machines have longer cycle length and richer transitions than the linear machines have. However, when evaluating the same attributes for conventional Geffe generators and their modified machines, we discover the new machines don't provide better results in general.

6.2 Future Work

There are many interesting aspects of this research that should be further studied. The most important ones include the following:

- Study the randomness property of the test patterns generated by the modified Geffe generators. This can provide evidence which may explain why the modified machines can achieve good fault coverage but with shorter cycle length and a smaller number of pattern pairs.

- Discover "good" LFSRs for the modified Geffe generators, so that the test sequences generated by these machines can have a longer cycle length and more transition pairs, and consequently, achieve higher fault coverage. There exist two solutions to this problem: using larger degree LFSRs, and searching for suitable LFSRs such that they and their embedded LFSRs are good matches. We realize that increasing the degree of the LFSR is not really feasible since, on the one hand, this can raise the number of flip-flops used in the new machines, and on the other hand, our research indicates modified Geffe generators with larger degree LFSRs do not necessarily produce improved quality test patterns.

- Verify whether different permutations of the test patterns generated by the modified Geffe generators can lead to better fault coverage.

- Investigate the performance of modified Geffe generators as a TPG to detect delay faults in digital testing. Similar to stuck-at faults in sequential circuits, a delay fault requires more than one pattern to detect. Therefore, the number of pattern transitions also plays an important role of affecting the delay fault detection ability of a TPG. It would be very interesting to investigate as to how

well the modified machines perform in testing circuits with delay faults presenting.

Bibliography

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

[2] V. D. Agrawal, C. R. Kime, and K. K. Saluja, *A Tutorial on Built-In Self-Test,* Design & Test of Computers, IEEE , Vol. 10, pp.73 – 82, March 1993

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

[4] F. Berglez, D. Bryan, and K. Kozminski, *Combinational Profiles of Sequential Benchmark Circuits*, Proc. IEEE Int. Symposium on Circuits and System, pp. 1929-1934, 1989.

[5] F. Berglez, D. Bryan, and K. Kozminski, *Notes on the ISCAS'89 Benchmark Circuits*, MCNC, 1989.

[6] J. Calver, *Analysis of Popular Benchmark Circuits,* M. Sc. Thesis, Department of Computer Science, University of Victoria, 2003.

[7] D. Densmore, *Built-In-Self Test (BIST) Implementations an Overview of Design Tradeoffs*, Technical paper, University of Michigan, 2001. [Online]. Available: http://www.cs.berkeley.edu/~densmore/documents/BIST.pdf

[8] P. R. Geffe, *How to Protect Data With Cipher That Are Really Hard to Break, Electronics*, Vol. 46, pp. 99-101, 1973.

[9] D. Kagaris and S. Tragoudas, *A Class of Good Characteristic Polynomials for LFSR Test Pattern Generator*, IEEE International Conference, pp. 292 – 295, Oct. 10, 1994.

[10] D. Kagaris and S. Tragoudas, *Cost-effective LFSR Synthesis for Optimal Pseudoxhaustive BIST Test Set*, IEEE Trans. on Very Large Scale Integration (VLSI) Systems, Vol. 1, pp. 526-536, Dec.1993.

[11] H. K. Lee and D. S. Ha, *Hope: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits*, Trans. on Computer Aided Design of Integrated Circuits and System, Vol. 15, pp.1048-1058, 1996.

[12] G. Marven, *Entropy Based Evaluation of Binary Sequences Produced by ALFSRs*, M. Sc. Thesis, Department of Computer Science, University of Victoria, 1994.

[13] G. Mrugalski, and J. Rajski, and J. Tyszer, *Cellular Automata-based Test Pattern Generators With Phase Shifters,* IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol. 19, pp. 878 – 893, Aug. 2000

[14] J. C. Muzio and M. Serra, *Quantitative Evaluation*, Fault tolerant digital system lecture slides, Feb. 2004.

[15] T. M. Niermann, W. T. Cheng, J. H. Patel, *Proofs: A Fast, Memory Efficient Sequential Circuit Fault Simulator,* Proc. ACM/IEEE Design Automation Conference, pp. 535-540, 1990.

[16] D. Qi, *A Comparison of Two Approaches to Geffe Test Pattern Generation*, Technical paper, Department of Computer Science, University of Victoria, 2004

[17] F. Ruskey, *Information on Primitive and Irreducible Polynomials*, Department of Computer Science, University of Victoria, 2000. [Online]. Available: http://www.theory.csc.uvic.ca/~cos/inf/neck/PolyInfo.html

[18] M. Serra and G. L. Chen, *Pseudo-Random Pattern Generation and Fault Coverage of Delay Faults with Non Linear Finite State Machines with High Entropy*, Proc. IEEE On-line Testing Workshop, 1997.

[19] H. S. Stone, Discrete Mathematical Structures and Their Application, Science Research Associates, Inc., 1973

[20] L. Y. Ungar, T. Ambler, *Economics of Built-In Self-Test,* Design & Test of Computers, IEEE , Vol. 18, pp. 70 – 79, Sept.-Oct. 2001

[21] Y. Z. Yu. *Delay Fault Coverage and the Entropy Based Cryptographic Strength on Binary Sequence Produced by Geffe Generators and ALFSMs*, M. Sc. Thesis, Department of Computer Science, University of Victoria, 2004.

[22] J. Zhong, *Non-linear Machines as Pseudo Random Pattern Generators for Digital Testing*, M. Sc. Thesis, Department of Computer Science, University of Victoria, 2003.

[23] S. Zhang, R. Byrne, J. C. Muzio, and D. M. Miller, *Quantitative Analysis for Linear Hybrid Cellular Automata and LFSR as Build-In Self-Test Generators for Sequential Faults*, Journal of Electronic Testing: Theory and Applications, Vol. &, pp. 2009-221, 1995.

Appendix 1: A Table of Primitive Characteristic Polynomials:

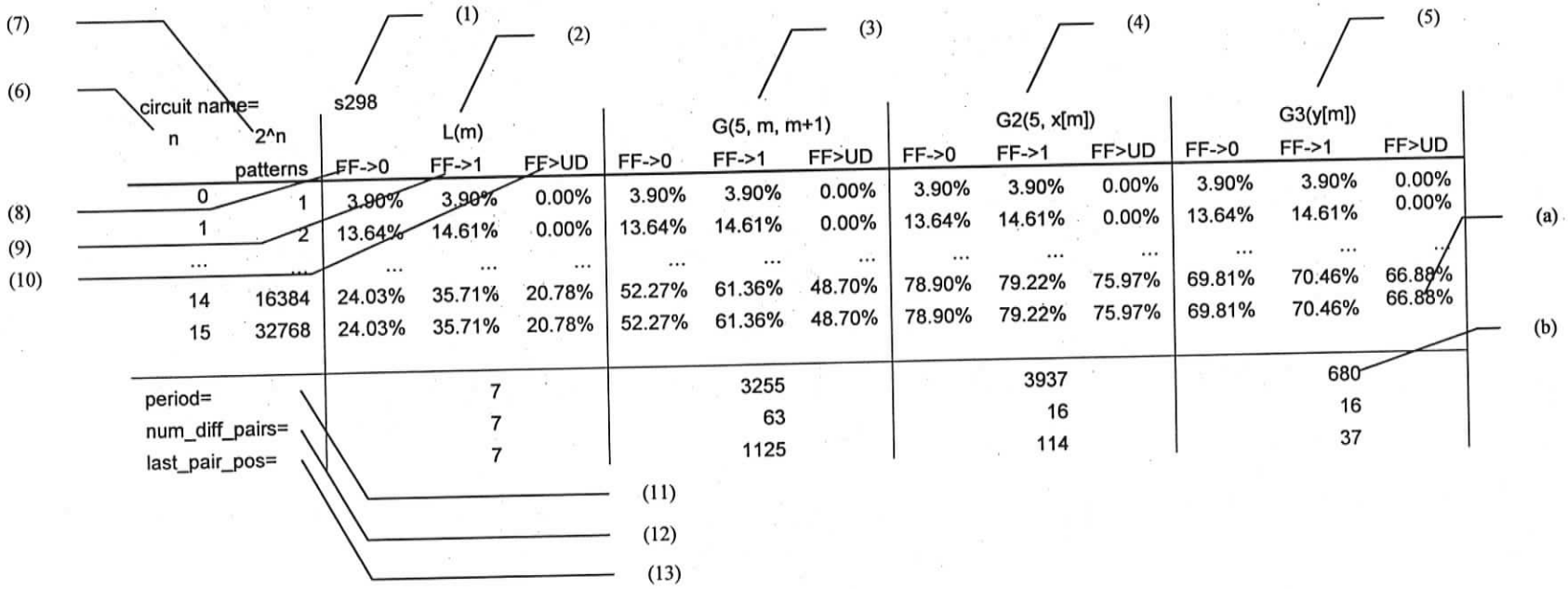
This appendix provides descriptions on the implementation of the test pattern generators examined by our experiments. The characteristic polynomials associated with all these generators are listed in the table below. These polynomials are primitive polynomials of degree n, $3 \leq n \leq 25$. A shorthand notation is employed. For example, the polynomial $x^7+x^6+x^3+x+1$ is represented by the entry, 7 6 3 1 0, listing the exponents of those x^i associated with a "1" coefficient. The polynomial with an underscore highlighting part of the function indicates it has a primitive polynomial embedding, and the embedded polynomial is the highlighted section. For example, polynomial 13 10 7 6 3 1 0 contains an embedded primitive polynomial 7 6 3 1 0.

# of inputs of CUT	LFSR		Conventional Geffe Generator		Modified Geffe Generator 2		Modified Geffe Generator 3	
	Notation	LFSR	Notation	LFSRs	Notation	LFSRs	Notation	$LFSR_4^+$
3	L(3)	3 1 0	G(5,3,4)	$LFSR_0$:5 2 0 $LFSR_1$:3 2 0 $LFSR_2$:4 1 0	$G_2(5,7[3])$	$LFSR_0$:5 2 0 $LFSR_3^+$:7 4 3 2 0	$G_3(7[3])$	10 5 3 2 0
4	L(4)	4 1 0	G(5,4,5)	$LFSR_0$:5 2 0 $LFSR_1$:4 3 0 $LFSR_2$:5 2 0	$G_2(5,7[4])$	$LFSR_0$:5 2 0 $LFSR_3^+$:7 5 4 3 0	$G_3(9[4])$	9 6 4 3 0
7	L(7)	7 1 0	G(5,7,8)	$LFSR_0$:5 2 0 $LFSR_1$:7 6 3 1 0 $LFSR_2$:8 6 5 1 0	$G_2(5,9[7])$	$LFSR_0$:5 2 0 $LFSR_3^+$:9 8 7 6 3 1 0	$G_3(13[7])$	13 10 7 6 3 1 0
8	L(8)	8 6 5 1 0	G(5,8,9)	$LFSR_0$:5 2 0 $LFSR_1$:8 4 3 2 0 $LFSR_2$:9 4 0	$G_2(5,11[8])$	$LFSR_0$: 5 2 0 $LFSR_3^+$:11 10 8 4 3 2 0	$G_3(15[8])$	15 10 8 4 3 2 0
9	L(9)	9 4 0	G(5,9,10)	$LFSR_0$:5 2 0 $LFSR_1$:9 8 7 6 3 2 0	$G_2(5,11[9])$	$LFSR_0$:5 2 0 $LFSR_3^+$:11 10 9 8 7 6 3 2	$G_3(16[9])$	16 15 14 13 12 11 9 8 7 6 3 2 0

				$LFSR_2$:10 3 0		0		
11	L(11)	11 2 0	G(5,11,12)	$LFSR_0$:5 2 0 $LFSR_1$:11 10 8 6 0 $LFSR_2$:12 7 4 3 0	$G_2(5,13[11])$	$LFSR_0$:5 2 0 $LFSR_3^+$:13 12 11 10 8 6 0	$G_3(17[11])$	17 16 11 10 8 6 0
12	L(12)	12 7 4 3 0	G(5,12,13)	$LFSR_0$:5 2 0 $LFSR_1$:12 11 8 6 0 $LFSR_2$:13 4 3 1 0	$G_2(5,17[12])$	$LFSR_0$:5 2 0 $LFSR_3^+$:17 15 12 11 8 6 0	$G_3(19[12])$	19 18 12 11 8 6 0
14	L(14)	14 12 11 1 0	G(5,14,15)	$LFSR_0$:5 2 0 $LFSR_1$:14 13 11 10 9 8 0 $LFSR_2$:15 1 0	$G_2(5,17[14])$	$LFSR_0$:5 2 0 $LFSR_3^+$:17 16 14 13 11 10 9 8 0	$G_3(23[14])$	23 22 21 19 14 13 11 10 9 8 0
16	L(16)	16 5 3 2 0	G(5,16,17)	$LFSR_0$:5 2 0 $LFSR_1$:16 5 3 2 0 $LFSR_2$:17 3 0	$G_2(5,19[16])$	$LFSR_0$:5 2 0 $LFSR_3^+$:19 17 16 5 3 2 0	$G_3(23[16])$	23 17 16 5 3 2 0
17	L(17)	17 3 0	G(5,17,18)	$LFSR_0$:5 2 0 $LFSR_1$:17 16 15 14 0 $LFSR_2$:18 7 0	$G_2(5,19[17])$	$LFSR_0$:5 2 0 $LFSR_3^+$:19 18 17 16 15 14 0	$G_3(23[17])$	23 21 20 18 17 16 15 14 0
18	L(18)	18 7 0	G(5,18,19)	$LFSR_0$:5 2 0 $LFSR_1$:18 17 15 14 13 12 0 $LFSR_2$:19 6 5 1 0	$G_2(5,23[18])$	$LFSR_0$:5 2 0 $LFSR_3^+$:23 22 18 17 15 14 13 12 0	$G_3(25[18])$	25 23 18 17 15 14 13 12 0
19	L(19)	19 6 5 1 0	G(5,19,20)	$LFSR_0$:5 2 0 $LFSR_1$:19 18 17 13 0 $LFSR_2$:20 3 0	$G_2(5,21[19])$	$LFSR_0$:5 2 0 $LFSR_3^+$:21 20 19 18 17 13 0	$G_3(25[19])$	25 23 19 18 17 13 0

Appendix 2: Experimental Results

This appendix provides the results of the two primary experiments: the fault simulation, and the test of the cycle length and transitions of test patterns. Each table is dedicated to the experimental results obtained by applying different test pattern generators to simulate an ISCAS'89 benchmark circuit. In total, there are 24 tables included in this appendix to present the test results from the circuits, which have less than 19 inputs. The following table template shows how these tables are constructed.



(1) –name of the circuit being simulated

(2) –notation of a LFSR. *m* is the degree of the LFSR, and *m* is equal to the number of inputs for the simulated circuit.

(3) –notation of a conventional Geffe generator. "5" is the degree of the selector LFSR, and m and m+1 are the degree of the two source LFSRs.

(4) –notation of a version 2 modified Geffe generator, where x is the degree of $LFSR_3^+$.

(5) –notation of a version 3 modified Geffe generator, where y is the degree of $LFSR_4^+$.

(6) –log2(number of test patterns).

(7) –number of test patterns used in the simulation.

(8) –notation, which indicates flip-flops of the circuit are initially set to logic 0.

(9) – notation, which indicates flip-flops of the circuit are initially set to logic 1.

(10) – notation, which indicates flip-lops of the circuit are initially set to undetermined.

(11) –the cycle length or period of the test sequence of 32768 patterns.

(12) –number of different transitions in the test sequence of 32768 patterns.

(13) –the index location of the last transition.

The percentages displayed in the table are a circuit's fault coverage collected by using different length test sequences produced by different type of generators. For example, (a) points the percentage of 66.88%. This fault coverage is the result of the fault simulation of circuit s298. In this simulation, all flip-flops of s298 are pre-set to the undetermined state, and the test sequence contains 32768 test patterns generated by a version 3 modified Geffe generator.

The bottom three rows of the table are results from the test of the cycle length and transitions of test patterns used in the fault simulation. For instance, (b) shows the version 3 modified Geffe generator produces a test sequence, which is able to cycle though 680 different states.

circuit name=		s298											
n	2^n patterns	L(3) FF->0	L(3) FF->1	L(3) FF->UD	G(5, 3, 4) FF->0	G(5, 3, 4) FF->1	G(5, 3, 4) FF->UD	G2(5, 7[3]) FF->0	G2(5, 7[3]) FF->1	G2(5, 7[3]) FF->UD	G3(7[3]) FF->0	G3(7[3]) FF->1	G3(7[3]) FF->UD
0	1	3.90%	3.90%	0.00%	3.90%	3.90%	0.00%	3.90%	3.90%	0.00%	3.90%	3.90%	0.00%
1	2	13.64%	14.61%	0.00%	13.64%	14.61%	0.00%	13.64%	14.61%	0.00%	13.64%	14.61%	0.00%
2	4	17.86%	28.25%	12.66%	16.88%	28.25%	12.66%	16.88%	28.25%	12.66%	16.88%	28.25%	12.66%
3	8	19.48%	31.82%	16.56%	22.73%	34.74%	19.48%	21.75%	35.07%	19.81%	22.73%	36.04%	20.46%
4	16	24.03%	35.71%	20.78%	26.95%	38.96%	23.70%	28.90%	40.26%	25.65%	23.38%	36.04%	20.78%
5	32	24.03%	35.71%	20.78%	26.95%	38.96%	23.70%	28.90%	40.26%	25.65%	33.77%	45.13%	30.52%
6	64	24.03%	35.71%	20.78%	26.95%	38.96%	23.70%	28.90%	40.26%	25.65%	34.09%	45.46%	30.52%
7	128	24.03%	35.71%	20.78%	38.96%	49.68%	35.07%	41.23%	52.27%	37.01%	50.00%	59.74%	47.08%
8	256	24.03%	35.71%	20.78%	38.96%	49.68%	35.07%	69.48%	70.13%	66.56%	56.49%	65.26%	52.92%
9	512	24.03%	35.71%	20.78%	38.96%	50.33%	35.07%	74.68%	75.00%	71.75%	69.81%	70.46%	66.88%
10	1024	24.03%	35.71%	20.78%	43.51%	54.55%	39.29%	75.00%	75.33%	72.08%	69.81%	70.46%	66.88%
11	2048	24.03%	35.71%	20.78%	52.27%	61.36%	48.70%	75.00%	75.33%	72.08%	69.81%	70.46%	66.88%
12	4096	24.03%	35.71%	20.78%	52.27%	61.36%	48.70%	78.90%	79.22%	75.97%	69.81%	70.46%	66.88%
13	8192	24.03%	35.71%	20.78%	52.27%	61.36%	48.70%	78.90%	79.22%	75.97%	69.81%	70.46%	66.88%
14	16384	24.03%	35.71%	20.78%	52.27%	61.36%	48.70%	78.90%	79.22%	75.97%	69.81%	70.46%	66.88%
15	32768	24.03%	35.71%	20.78%	52.27%	61.36%	48.70%	78.90%	79.22%	75.97%	69.81%	70.46%	66.88%
period=			7			3255			3937			680	
num_diff_pairs=			7			63			16			16	
last_pair_pos=			7			1125			114			37	

circuit name=		s382											
n	2^n	L(3)			G(5, 3, 4)			G2(5, 7[3])			G3(7[3])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	1.50%	1.50%	0.00%	1.50%	1.50%	0.00%	1.50%	1.50%	0.00%	1.50%	1.50%	0.00%
1	2	6.77%	6.27%	3.76%	10.53%	7.77%	0.00%	10.53%	7.77%	0.00%	10.53%	7.77%	0.00%
2	4	11.28%	13.03%	10.28%	10.53%	12.28%	5.76%	11.28%	16.79%	10.28%	10.53%	12.28%	5.76%
3	8	12.78%	14.54%	11.78%	11.28%	16.79%	10.28%	11.28%	16.79%	10.28%	11.28%	16.79%	10.28%
4	16	12.78%	14.54%	11.78%	12.78%	18.30%	11.78%	12.78%	18.30%	11.78%	12.78%	18.30%	11.78%
5	32	12.78%	14.54%	11.78%	12.78%	18.30%	11.78%	13.28%	18.80%	12.28%	13.28%	18.80%	12.28%
6	64	13.03%	14.54%	11.78%	13.03%	18.30%	11.78%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%
7	128	13.03%	14.54%	11.78%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%
8	256	13.03%	14.54%	11.78%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%
9	512	13.03%	14.54%	11.78%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%
10	1024	13.03%	14.54%	11.78%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%
11	2048	13.03%	14.54%	11.78%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%
12	4096	13.03%	14.54%	11.78%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%
13	8192	13.03%	14.54%	11.78%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%
14	16384	13.03%	14.54%	11.78%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%
15	32768	13.03%	14.54%	11.78%	13.53%	18.80%	12.28%	13.53%	18.80%	12.28%			
period=			7			3255			3937			680	
num_diff_pairs=			7			63			16			16	
last_pair_pos=			7			1125			114			37	

circuit name=		s400											
n	2^n	L(3)			G(5, 3, 4)			G2(5, 7[3])			G3(7[3])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	1.42%	1.42%	0.00%	1.42%	1.42%	0.00%	1.42%	1.42%	0.00%	1.42%	1.42%	0.00%
1	2	5.66%	6.37%	3.77%	9.43%	7.31%	0.00%	9.43%	7.31%	0.00%	9.43%	7.31%	0.00%
2	4	10.61%	12.26%	9.43%	9.43%	11.09%	4.72%	10.61%	16.04%	9.43%	9.43%	11.09%	4.72%
3	8	12.26%	14.15%	11.32%	10.61%	16.04%	9.67%	10.61%	16.04%	9.67%	10.61%	16.04%	9.67%
4	16	12.26%	14.15%	11.32%	12.50%	17.93%	11.56%	12.26%	17.69%	11.32%	12.26%	17.69%	11.32%
5	32	12.26%	14.15%	11.32%	12.50%	17.93%	11.56%	12.97%	18.40%	12.03%	12.97%	18.40%	12.03%
6	64	12.50%	14.15%	11.32%	12.74%	17.93%	11.56%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%
7	128	12.50%	14.15%	11.32%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%
8	256	12.50%	14.15%	11.32%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%
9	512	12.50%	14.15%	11.32%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%
10	1024	12.50%	14.15%	11.32%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%
11	2048	12.50%	14.15%	11.32%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%
12	4096	12.50%	14.15%	11.32%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%
13	8192	12.50%	14.15%	11.32%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%
14	16384	12.50%	14.15%	11.32%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%
15	32768	12.50%	14.15%	11.32%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%	13.21%	18.40%	12.03%
period=			7			3255			3937			680	
num_diff_pairs=			7			63			16			16	
last_pair_pos=			7			1125			114			37	

circuit name=		s444											
n	2^n	L(3)			G(5, 3, 4)			G2(5, 7[3])			G3(7[3])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	1.27%	1.27%	0.00%	1.27%	1.27%	0.00%	1.27%	1.27%	0.00%	1.27%	1.27%	0.00%
1	2	8.44%	6.54%	0.00%	8.44%	6.54%	0.00%	8.44%	6.54%	0.00%	8.44%	6.54%	0.00%
2	4	9.49%	14.35%	8.44%	8.44%	9.92%	4.22%	9.49%	14.35%	8.44%	8.44%	9.92%	4.22%
3	8	9.49%	14.35%	8.65%	11.39%	16.25%	10.55%	11.18%	16.03%	10.34%	11.39%	16.25%	10.55%
4	16	11.18%	16.03%	10.34%	11.39%	16.25%	10.55%	11.39%	16.25%	10.55%	11.39%	16.25%	10.55%
5	32	11.18%	16.03%	10.34%	11.39%	16.25%	10.55%	11.39%	16.25%	10.55%	12.03%	16.88%	11.18%
6	64	11.39%	16.03%	10.34%	11.60%	16.25%	10.55%	11.60%	16.25%	10.55%	12.24%	16.88%	11.18%
7	128	11.39%	16.03%	10.34%	12.24%	16.88%	11.18%	12.24%	16.88%	11.18%	12.24%	16.88%	11.18%
8	256	11.39%	16.03%	10.34%	12.24%	16.88%	11.18%	17.30%	21.73%	16.25%	13.50%	18.14%	12.45%
9	512	11.39%	16.03%	10.34%	12.24%	16.88%	11.18%	17.30%	21.73%	16.25%	17.30%	21.73%	16.25%
10	1024	11.39%	16.03%	10.34%	12.24%	16.88%	11.18%	17.30%	21.73%	16.25%	17.30%	21.73%	16.25%
11	2048	11.39%	16.03%	10.34%	12.24%	16.88%	11.18%	17.30%	21.73%	16.25%	17.30%	21.73%	16.25%
12	4096	11.39%	16.03%	10.34%	12.24%	16.88%	11.18%	19.62%	24.05%	18.57%	17.30%	21.73%	16.25%
13	8192	11.39%	16.03%	10.34%	12.24%	16.88%	11.18%	19.62%	24.05%	18.57%	17.30%	21.73%	16.25%
14	16384	11.39%	16.03%	10.34%	12.24%	16.88%	11.18%	19.62%	24.05%	18.57%	17.30%	21.73%	16.25%
15	32768	11.39%	16.03%	10.34%	12.24%	16.88%	11.18%	19.62%	24.05%	18.57%	17.30%	21.73%	16.25%
period=			7			3255			3937			680	
num_diff_pairs=			7			63			16			16	
last_pair_pos=			7			1125			114			37	

circuit name=		s526n											
n	2^n	L(3)			G(5, 3, 4)			G2(5, 7[3])			G3(7[3])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	2.17%	2.17%	0.00%	2.17%	2.17%	0.00%	2.17%	2.17%	0.00%	2.17%	2.17%	0.00%
1	2	7.60%	8.14%	0.00%	7.60%	8.14%	0.00%	7.60%	8.14%	0.00%	7.60%	8.14%	0.00%
2	4	9.58%	16.46%	7.05%	9.04%	16.46%	7.05%	9.04%	16.46%	7.05%	9.04%	16.46%	7.05%
3	8	9.58%	17.90%	8.50%	9.58%	17.90%	8.50%	9.58%	17.90%	8.50%	9.58%	17.90%	8.50%
4	16	9.58%	17.90%	8.50%	9.58%	17.90%	8.50%	9.77%	18.08%	8.68%	9.77%	18.08%	8.68%
5	32	9.58%	17.90%	8.50%	9.58%	17.90%	8.50%	9.77%	18.08%	8.68%	9.95%	18.08%	8.68%
6	64	9.77%	17.90%	8.50%	9.77%	17.90%	8.50%	9.95%	18.08%	8.68%	10.67%	18.81%	9.40%
7	128	9.77%	17.90%	8.50%	9.95%	18.08%	8.68%	13.56%	21.88%	12.30%	10.67%	18.81%	9.40%
8	256	9.77%	17.90%	8.50%	9.95%	18.08%	8.68%	13.56%	21.88%	12.30%	13.56%	21.88%	12.30%
9	512	9.77%	17.90%	8.50%	9.95%	18.08%	8.68%	13.56%	21.88%	12.30%	13.56%	21.88%	12.30%
10	1024	9.77%	17.90%	8.50%	9.95%	18.08%	8.68%	14.29%	22.79%	13.02%	13.56%	21.88%	12.30%
11	2048	9.77%	17.90%	8.50%	9.95%	18.08%	8.68%	15.01%	23.33%	13.56%	13.56%	21.88%	12.30%
12	4096	9.77%	17.90%	8.50%	9.95%	18.08%	8.68%	15.01%	23.33%	13.56%	13.56%	21.88%	12.30%
13	8192	9.77%	17.90%	8.50%	9.95%	18.08%	8.68%	15.01%	23.33%	13.56%	13.56%	21.88%	12.30%
14	16384	9.77%	17.90%	8.50%	9.95%	18.08%	8.68%	15.01%	23.33%	13.56%	13.56%	21.88%	12.30%
15	32768	9.77%	17.90%	8.50%									
period=			7			3255			3937			680	
num_diff_pairs=			7			63			16			16	
last_pair_pos=			7			1125			114			37	

circuit name=		s526											
n	2^n	L(3)			G(5, 3, 4)			G2(5, 7[3])			G3(7[3])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	2.16%	2.16%	0.00%	2.16%	2.16%	0.00%	2.16%	2.16%	0.00%	2.16%	2.16%	0.00%
1	2	7.57%	8.11%	0.00%	7.57%	8.11%	0.00%	7.57%	8.11%	0.00%	7.57%	8.11%	0.00%
2	4	9.55%	16.40%	7.03%	9.01%	16.40%	7.03%	9.01%	16.40%	7.03%	9.01%	16.40%	7.03%
3	8	9.55%	17.84%	8.47%	9.55%	17.84%	8.47%	9.55%	17.84%	8.47%	9.55%	17.84%	8.47%
4	16	9.55%	17.84%	8.47%	9.55%	17.84%	8.47%	9.73%	18.02%	8.65%	9.55%	17.84%	8.47%
5	32	9.55%	17.84%	8.47%	9.55%	17.84%	8.47%	9.73%	18.02%	8.65%	9.73%	18.02%	8.65%
6	64	9.73%	17.84%	8.47%	9.73%	17.84%	8.47%	9.91%	18.02%	8.65%	9.91%	18.02%	8.65%
7	128	9.73%	17.84%	8.47%	9.91%	18.02%	8.65%	9.91%	18.02%	8.65%	10.63%	18.74%	9.37%
8	256	9.73%	17.84%	8.47%	9.91%	18.02%	8.65%	13.51%	21.80%	12.25%	10.63%	18.74%	9.37%
9	512	9.73%	17.84%	8.47%	9.91%	18.02%	8.65%	13.51%	21.80%	12.25%	13.51%	21.80%	12.25%
10	1024	9.73%	17.84%	8.47%	9.91%	18.02%	8.65%	13.51%	21.80%	12.25%	13.51%	21.80%	12.25%
11	2048	9.73%	17.84%	8.47%	9.91%	18.02%	8.65%	14.23%	22.70%	12.97%	13.51%	21.80%	12.25%
12	4096	9.73%	17.84%	8.47%	9.91%	18.02%	8.65%	14.96%	23.24%	13.51%	13.51%	21.80%	12.25%
13	8192	9.73%	17.84%	8.47%	9.91%	18.02%	8.65%	14.96%	23.24%	13.51%	13.51%	21.80%	12.25%
14	16384	9.73%	17.84%	8.47%	9.91%	18.02%	8.65%	14.96%	23.24%	13.51%	13.51%	21.80%	12.25%
15	32768	9.73%	17.84%	8.47%	9.91%	18.02%	8.65%	14.96%	23.24%	13.51%	13.51%	21.80%	12.25%
period=			7			3255			3937			680	
num_diff_pairs=			7			63			16			16	
last_pair_pos=			7			1125			114			37	

circuit name= s27

n	2^n patterns	L(4)			G(5, 4, 5)			G2(5, 7[4])			G3(9[4])		
		FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	15.63%	9.38%	0.00%	18.75%	9.38%	0.00%	18.75%	9.38%	0.00%	18.75%	9.38%	0.00%
1	2	34.38%	15.63%	12.50%	50.00%	28.13%	0.00%	50.00%	28.13%	0.00%	50.00%	28.13%	0.00%
2	4	50.00%	46.88%	43.75%	65.63%	28.13%	18.75%	87.50%	28.13%	25.00%	65.63%	28.13%	18.75%
3	8	59.38%	62.50%	59.38%	65.63%	28.13%	18.75%	90.63%	59.38%	56.25%	65.63%	31.25%	21.88%
4	16	71.88%	71.88%	71.88%	81.25%	71.88%	71.88%	90.63%	78.13%	75.00%	71.88%	34.38%	31.25%
5	32	71.88%	71.88%	71.88%	96.88%	96.88%	96.88%	93.75%	96.88%	93.75%	87.50%	87.50%	84.38%
6	64	71.88%	71.88%	71.88%	100.00%	100.00%	100.00%	93.75%	96.88%	93.75%	87.50%	90.63%	87.50%
7	128	71.88%	71.88%	71.88%	100.00%	100.00%	100.00%	93.75%	96.88%	93.75%	87.50%	90.63%	87.50%
8	256	71.88%	71.88%	71.88%	100.00%	100.00%	100.00%	93.75%	96.88%	93.75%	87.50%	90.63%	87.50%
9	512	71.88%	71.88%	71.88%	100.00%	100.00%	100.00%	93.75%	96.88%	93.75%	87.50%	90.63%	87.50%
10	1024	71.88%	71.88%	71.88%	100.00%	100.00%	100.00%	93.75%	96.88%	93.75%	87.50%	90.63%	87.50%
11	2048	71.88%	71.88%	71.88%	100.00%	100.00%	100.00%	93.75%	96.88%	93.75%	87.50%	90.63%	87.50%
12	4096	71.88%	71.88%	71.88%	100.00%	100.00%	100.00%	93.75%	96.88%	93.75%	87.50%	90.63%	87.50%
13	8192	71.88%	71.88%	71.88%	100.00%	100.00%	100.00%	93.75%	96.88%	93.75%	87.50%	90.63%	87.50%
14	16384	71.88%	71.88%	71.88%	100.00%	100.00%	100.00%	93.75%	96.88%	93.75%	87.50%	90.63%	87.50%
15	32768	71.88%	71.88%	71.88%	100.00%	100.00%	100.00%	93.75%	96.88%	93.75%	87.50%	90.63%	87.50%
period=			15			465			3937			121	
num_diff_pairs=			15			207			32			30	
last_pair_pos=			15			465			205			107	

circuit name=		s386											
n	2^n	L(7)			G(5, 7, 8)			G2(5, 9[7])			G3(13[7])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	9.64%	1.82%	0.52%	4.17%	1.82%	1.56%	4.17%	1.82%	1.56%	4.17%	1.82%	1.56%
1	2	13.02%	4.95%	1.82%	7.81%	8.59%	1.56%	7.81%	8.59%	1.56%	7.81%	8.59%	1.56%
2	4	20.83%	14.06%	3.39%	19.79%	21.09%	14.06%	18.49%	19.79%	12.76%	19.79%	21.09%	14.06%
3	8	28.39%	22.40%	7.29%	28.91%	30.99%	24.74%	31.25%	33.85%	27.60%	27.08%	30.21%	23.96%
4	16	29.69%	25.78%	25.78%	37.76%	39.84%	36.72%	33.33%	36.46%	30.73%	31.77%	35.16%	30.73%
5	32	32.81%	29.17%	29.17%	41.41%	43.49%	40.63%	42.97%	42.97%	42.97%	36.20%	36.72%	35.94%
6	64	34.12%	30.47%	30.47%	46.35%	48.44%	45.57%	45.57%	45.57%	45.57%	44.01%	44.53%	43.75%
7	128	34.12%	30.47%	30.47%	50.78%	51.04%	50.78%	46.88%	46.88%	46.88%	45.31%	45.83%	45.31%
8	256	34.12%	30.47%	30.47%	52.34%	52.60%	52.34%	47.14%	47.14%	47.14%	46.88%	46.88%	46.88%
9	512	34.12%	30.47%	30.47%	53.39%	53.39%	53.39%	47.40%	47.40%	47.40%	47.14%	47.14%	47.14%
10	1024	34.12%	30.47%	30.47%	58.07%	58.07%	58.07%	47.40%	47.40%	47.40%	47.40%	47.40%	47.40%
11	2048	34.12%	30.47%	30.47%	60.16%	60.16%	60.16%	47.40%	47.40%	47.40%	47.40%	47.40%	47.40%
12	4096	34.12%	30.47%	30.47%	61.46%	61.46%	61.46%	47.40%	47.40%	47.40%	47.40%	47.40%	47.40%
13	8192	34.12%	30.47%	30.47%	67.97%	67.97%	67.97%	47.40%	47.40%	47.40%	47.40%	47.40%	47.40%
14	16384	34.12%	30.47%	30.47%	68.23%	68.23%	68.23%	47.40%	47.40%	47.40%	47.40%	47.40%	47.40%
15	32768	34.12%	30.47%	30.47%	69.27%	69.27%	69.27%						
period=			127			>32768			11811			710	
num_diff_pairs=			127			11456			256			220	
last_pair_pos=			127			32763			2239			693	

circuit name=		s1488											
n	2^n	L(8)			G(5, 8, 9)			G2(5,11[8])			G3(15[8])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	8.88%	5.05%	0.47%	8.75%	5.05%	0.47%	8.75%	5.05%	0.47%	8.75%	5.05%	0.47%
1	2	9.89%	12.99%	9.29%	12.32%	12.58%	8.55%	12.32%	12.58%	8.55%	12.32%	12.58%	8.55%
2	4	17.97%	21.20%	17.83%	21.13%	23.15%	20.32%	24.76%	26.45%	23.49%	21.53%	23.55%	20.73%
3	8	30.96%	33.51%	30.69%	24.97%	27.66%	24.83%	30.22%	33.04%	29.95%	22.28%	24.90%	22.07%
4	16	32.03%	34.72%	31.90%	25.84%	28.53%	25.57%	32.30%	34.86%	32.10%	25.57%	28.26%	25.51%
5	32	35.40%	37.95%	35.20%	31.09%	33.65%	30.89%	33.98%	36.54%	33.85%	31.63%	34.19%	31.43%
6	64	47.58%	49.06%	47.38%	41.12%	43.27%	40.92%	35.53%	38.09%	35.33%	34.05%	36.61%	33.85%
7	128	48.05%	49.53%	47.85%	42.80%	44.95%	42.67%	38.43%	40.98%	38.22%	46.23%	47.91%	46.03%
8	256	48.05%	49.53%	47.85%	48.99%	50.67%	48.86%	42.67%	44.82%	42.46%	46.64%	48.32%	46.43%
9	512	48.05%	49.53%	47.85%	49.13%	50.81%	48.99%	43.07%	45.22%	42.87%	46.70%	48.39%	46.50%
10	1024	48.05%	49.53%	47.85%	54.71%	55.86%	54.58%	43.07%	45.22%	42.87%	53.23%	54.51%	53.03%
11	2048	48.05%	49.53%	47.85%	56.73%	57.87%	56.60%	53.23%	54.51%	53.03%	53.23%	54.51%	53.03%
12	4096	48.05%	49.53%	47.85%	57.87%	59.02%	57.74%	54.71%	55.86%	54.51%	53.23%	54.51%	53.03%
13	8192	48.05%	49.53%	47.85%	62.92%	64.07%	62.79%	56.06%	57.20%	55.86%	54.71%	55.86%	54.51%
14	16384	48.05%	49.53%	47.85%	67.03%	68.17%	66.89%	56.06%	57.20%	55.86%	54.71%	55.86%	54.51%
15	32768	48.05%	49.53%	47.85%	67.83%	68.84%	67.70%	56.06%	57.20%	55.86%	54.71%	55.86%	54.51%
period=			255			>32768			19685			24247	
num_diff_pairs=			255			15788			512			512	
last_pair_pos=			255			32767			3864			5760	

circuit name=		s1494											
n	2^n	L(8)			G(5, 8, 9)			G2(5,11[8])			G3(15[8])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	8.77%	4.98%	0.47%	8.63%	4.98%	0.47%	8.63%	4.98%	0.47%	8.63%	4.98%	0.47%
1	2	9.76%	12.82%	9.16%	12.15%	12.42%	8.43%	12.15%	12.42%	8.43%	12.15%	12.42%	8.43%
2	4	17.80%	20.98%	17.66%	20.92%	22.91%	20.12%	24.24%	25.90%	22.98%	21.25%	23.24%	20.45%
3	8	30.35%	32.87%	30.08%	24.64%	27.29%	24.50%	29.68%	32.47%	29.42%	21.98%	24.57%	21.78%
4	16	31.41%	34.06%	31.28%	25.50%	28.15%	25.23%	31.67%	34.20%	31.47%	25.23%	27.89%	25.17%
5	32	34.66%	37.19%	34.46%	30.48%	33.00%	30.28%	33.27%	35.79%	33.13%	30.94%	33.47%	30.74%
6	64	46.95%	48.41%	46.75%	40.31%	42.43%	40.11%	34.79%	37.32%	34.60%	33.33%	35.86%	33.13%
7	128	47.41%	48.87%	47.21%	41.97%	44.09%	41.83%	37.65%	40.17%	37.45%	45.42%	47.08%	45.22%
8	256	47.41%	48.87%	47.21%	48.14%	49.80%	48.01%	41.83%	43.96%	41.63%	45.82%	47.48%	45.62%
9	512	47.41%	48.87%	47.21%	48.27%	49.93%	48.14%	42.23%	44.36%	42.03%	45.88%	47.54%	45.68%
10	1024	47.41%	48.87%	47.21%	53.79%	54.91%	53.65%	42.23%	44.36%	42.03%	52.26%	53.52%	52.06%
11	2048	47.41%	48.87%	47.21%	55.78%	56.91%	55.64%	52.26%	53.52%	52.06%	52.26%	53.52%	52.06%
12	4096	47.41%	48.87%	47.21%	56.91%	58.04%	56.77%	53.92%	55.05%	53.72%	52.26%	53.52%	52.06%
13	8192	47.41%	48.87%	47.21%	61.89%	63.02%	61.75%	55.31%	56.44%	55.11%	53.92%	55.05%	53.72%
14	16384	47.41%	48.87%	47.21%	66.20%	67.33%	66.07%	55.31%	56.44%	55.11%	53.92%	55.05%	53.72%
15	32768	47.41%	48.87%	47.21%	67.00%	68.00%	66.87%	55.31%	56.44%	55.11%	53.92%	55.05%	53.72%
period=			255			>32768			19685			24247	
num_diff_pairs=			255			15788			512			512	
last_pair_pos=			255			32767			3864			5760	

circuit name= s344

n	2^n patterns	L(9) FF->0	L(9) FF->1	L(9) FF->UD	G(5, 9, 10) FF->0	G(5, 9, 10) FF->1	G(5, 9, 10) FF->UD	G2(5, 11[9]) FF->0	G2(5, 11[9]) FF->1	G2(5, 11[9]) FF->UD	G3(16[9]) FF->0	G3(16[9]) FF->1	G3(16[9]) FF->UD
0	1	6.14%	7.90%	0.00%	6.14%	7.90%	0.00%	6.14%	7.90%	0.00%	6.14%	7.90%	0.00%
1	2	21.35%	19.88%	0.00%	21.35%	19.88%	0.00%	21.35%	19.88%	0.00%	21.35%	19.88%	0.00%
2	4	33.33%	33.92%	15.50%	35.09%	33.63%	11.11%	35.09%	33.63%	11.11%	35.09%	33.63%	11.11%
3	8	56.43%	51.17%	44.15%	56.43%	53.22%	45.61%	54.09%	54.09%	42.69%	67.54%	65.50%	57.60%
4	16	64.33%	60.53%	54.97%	71.93%	70.47%	66.37%	66.37%	67.25%	59.06%	82.46%	80.70%	76.02%
5	32	69.59%	68.42%	63.74%	78.07%	77.19%	73.68%	80.99%	80.70%	76.61%	86.84%	85.67%	81.87%
6	64	70.47%	69.01%	64.62%	78.66%	78.07%	74.85%	81.87%	81.29%	77.49%	92.11%	92.69%	89.47%
7	128	71.05%	69.59%	65.21%	90.64%	90.64%	87.72%	83.63%	82.75%	79.24%	92.11%	92.69%	89.47%
8	256	79.53%	77.78%	73.98%	92.11%	92.11%	89.18%	91.52%	91.23%	88.01%	93.57%	94.15%	90.94%
9	512	79.53%	77.78%	73.98%	95.32%	95.91%	92.98%	95.03%	94.74%	91.52%	93.57%	94.15%	90.94%
10	1024	79.53%	77.78%	73.98%	96.49%	97.37%	94.44%	95.03%	94.74%	91.52%	94.44%	95.03%	91.81%
11	2048	79.53%	77.78%	73.98%	97.08%	98.25%	95.32%	95.03%	95.32%	92.11%	95.61%	96.20%	92.98%
12	4096	79.53%	77.78%	73.98%	97.08%	98.25%	95.32%	96.78%	97.37%	94.15%	95.61%	96.20%	92.98%
13	8192	79.53%	77.78%	73.98%	97.37%	98.54%	95.61%	96.78%	97.37%	94.15%	95.61%	96.20%	92.98%
14	16384	79.53%	77.78%	73.98%	97.95%	99.12%	96.20%	96.78%	97.37%	94.15%	96.20%	96.78%	93.57%
15	32768	79.53%	77.78%	73.98%	97.95%	99.12%	96.20%	96.78%	97.37%	94.15%	96.20%	97.08%	93.86%
period=			511			>32768			31713			>32768	
num_diff_pairs=			511			17139			1024			1024	
last_pair_pos=			511			32767			13815			9856	

circuit name= s349

n	2^n patterns	L(9) FF->0	L(9) FF->1	L(9) FF->UD	G(5, 9, 10) FF->0	G(5, 9, 10) FF->1	G(5, 9, 10) FF->UD	G2(5, 11[9]) FF->0	G2(5, 11[9]) FF->1	G2(5, 11[9]) FF->UD	G3(16[9]) FF->0	G3(16[9]) FF->1	G3(16[9]) FF->UD
0	1	6.00%	7.71%	0.00%	6.00%	7.71%	0.00%	6.00%	7.71%	0.00%	6.00%	7.71%	0.00%
1	2	21.14%	19.43%	0.00%	21.14%	19.43%	0.00%	21.14%	19.43%	0.00%	21.14%	19.43%	0.00%
2	4	32.86%	34.00%	15.43%	35.14%	33.43%	10.86%	35.14%	33.43%	10.86%	35.14%	33.43%	10.86%
3	8	56.29%	51.14%	44.29%	56.29%	53.14%	45.71%	53.71%	54.00%	42.57%	67.14%	65.14%	57.43%
4	16	64.00%	60.29%	54.86%	71.43%	70.00%	66.00%	66.00%	66.86%	58.86%	81.71%	80.00%	75.43%
5	32	69.14%	68.00%	63.43%	77.43%	76.57%	73.14%	80.29%	80.00%	76.00%	86.00%	84.86%	81.14%
6	64	70.00%	68.57%	64.29%	78.00%	77.43%	74.29%	81.14%	80.57%	76.86%	91.71%	92.29%	89.14%
7	128	70.57%	69.14%	64.86%	90.29%	90.29%	87.43%	83.43%	82.57%	79.14%	91.71%	92.29%	89.14%
8	256	79.43%	77.71%	74.00%	91.71%	91.71%	88.86%	91.14%	90.86%	87.71%	93.14%	93.71%	90.57%
9	512	79.43%	77.71%	74.00%	94.86%	95.43%	92.57%	94.57%	94.29%	91.14%	93.14%	93.71%	90.57%
10	1024	79.43%	77.71%	74.00%	96.00%	96.86%	94.00%	94.57%	94.29%	91.14%	94.00%	94.57%	91.43%
11	2048	79.43%	77.71%	74.00%	96.57%	97.71%	94.86%	94.57%	94.86%	91.71%	95.14%	95.71%	92.57%
12	4096	79.43%	77.71%	74.00%	96.57%	97.71%	94.86%	96.29%	96.86%	93.71%	95.14%	95.71%	92.57%
13	8192	79.43%	77.71%	74.00%	96.86%	98.00%	95.14%	96.29%	96.86%	93.71%	95.14%	95.71%	92.57%
14	16384	79.43%	77.71%	74.00%	97.43%	98.57%	95.71%	96.29%	96.86%	93.71%	95.71%	96.29%	93.14%
15	32768	79.43%	77.71%	74.00%	97.43%	98.57%	95.71%	96.29%	96.86%	93.71%	95.71%	96.57%	93.43%

period=			511			>32768			31713			>32768	
num_diff_pairs=			511			17139			1024			1024	
last_pair_pos=			511			32767			13815			9856	

circuit name=		s208											
n	2^n	L(11)			G(5, 11, 12)			G2(5, 13[11])			G3(17[11])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	0.46%	0.46%	0.46%	0.46%	0.46%	0.46%	0.46%	0.46%	0.46%	0.46%	0.46%	0.46%
1	2	5.07%	1.84%	0.46%	7.83%	5.53%	0.46%	7.83%	5.53%	0.46%	7.83%	5.53%	0.46%
2	4	6.45%	5.07%	2.30%	11.52%	7.83%	3.69%	13.36%	8.76%	4.15%	10.14%	5.99%	1.38%
3	8	9.22%	7.37%	4.15%	15.21%	11.98%	6.45%	15.67%	11.98%	6.45%	17.51%	17.05%	5.99%
4	16	13.83%	11.98%	6.45%	20.74%	27.19%	6.45%	20.74%	13.36%	6.45%	23.96%	18.89%	5.99%
5	32	33.64%	22.58%	6.45%	35.48%	30.42%	6.91%	32.26%	25.35%	6.45%	30.88%	29.49%	6.45%
6	64	52.54%	44.70%	6.91%	38.71%	58.07%	6.91%	52.07%	55.30%	6.45%	45.16%	58.07%	6.45%
7	128	67.74%	66.36%	6.91%	56.68%	61.75%	6.91%	59.45%	73.73%	6.45%	71.43%	59.91%	7.83%
8	256	74.65%	70.51%	7.37%	73.73%	72.81%	6.91%	78.34%	81.11%	7.83%	75.58%	68.66%	8.30%
9	512	79.72%	82.95%	7.37%	90.32%	82.49%	6.91%	83.87%	91.24%	7.83%	84.79%	87.56%	8.30%
10	1024	94.93%	98.16%	7.37%	90.78%	94.01%	7.83%	93.09%	99.08%	7.83%	86.64%	93.55%	8.30%
11	2048	98.62%	99.08%	8.30%	94.47%	94.01%	8.30%	94.93%	99.08%	8.30%	92.17%	95.39%	8.30%
12	4096	98.62%	99.08%	8.30%	99.54%	100.00%	8.30%	99.54%	99.08%	8.30%	92.63%	99.54%	8.30%
13	8192	98.62%	99.08%	8.30%	100.00%	100.00%	8.30%	100.00%	99.54%	8.30%	100.00%	100.00%	8.30%
14	16384	98.62%	99.08%	8.30%	100.00%	100.00%	8.30%	100.00%	99.54%	8.30%	100.00%	100.00%	8.30%
15	32768	98.62%	99.08%	8.30%	100.00%	100.00%	8.30%	100.00%	100.00%	8.30%	100.00%	100.00%	8.30%
period=			2047			>32768			15810			21075	
num_diff_pairs=			2047			21781			3813			3945	
last_pair_pos=			2047			32767			15737			20947	

circuit name= s38584

n	2^n patterns	L(12) FF->0	L(12) FF->1	L(12) FF->UD	G(5, 12, 13) FF->0	G(5, 12, 13) FF->1	G(5, 12, 13) FF->UD	G2(5, 17[12]) FF->0	G2(5, 17[12]) FF->1	G2(5, 17[12]) FF->UD	G3(19[12]) FF->0	G3(19[12]) FF->1	G3(19[12]) FF->UD
0	1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
1	2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
2	4	1.98%	2.12%	0.49%	0.00%	0.00%	0.00%	1.85%	2.15%	0.41%	1.86%	2.19%	0.36%
3	8	3.11%	3.64%	1.06%	1.87%	2.31%	0.26%	3.02%	3.78%	0.97%	3.11%	3.85%	1.01%
4	16	3.95%	5.12%	1.57%	3.89%	4.91%	1.39%	3.93%	5.16%	1.58%	5.99%	6.96%	1.49%
5	32	10.37%	12.80%	3.08%	8.89%	9.87%	2.23%	10.92%	12.86%	3.06%	11.36%	13.78%	2.48%
6	64	16.35%	19.57%	6.20%	15.21%	16.56%	2.77%	18.12%	19.71%	5.18%	18.75%	20.45%	6.13%
7	128	27.27%	27.97%	14.26%	26.03%	25.88%	5.74%	28.22%	29.33%	12.46%	29.06%	28.84%	10.99%
8	256	35.72%	37.61%	28.68%	34.71%	35.59%	10.84%	35.78%	38.02%	21.40%	35.88%	38.06%	23.72%
9	512	41.43%	42.46%	37.63%	40.01%	41.18%	24.12%	41.26%	42.78%	31.14%	40.78%	43.03%	33.41%
10	1024	46.35%	46.89%	43.10%	44.86%	45.50%	36.11%	45.88%	47.43%	40.23%	45.12%	47.31%	40.76%
11	2048	50.92%	51.24%	47.93%	47.99%	48.87%	43.89%	50.86%	52.21%	46.88%	50.94%	52.29%	47.65%
12	4096	55.00%	55.30%	51.81%	50.91%	51.99%	48.58%	53.99%	55.00%	51.72%	53.81%	54.80%	51.26%
13	8192	58.10%	58.32%	56.36%	55.05%	56.17%	53.81%	58.00%	58.67%	56.78%	57.68%	58.61%	56.30%
14	16384	61.17%	61.33%	59.84%	59.30%	60.25%	58.49%	61.26%	61.95%	60.50%	61.14%	61.95%	60.29%
15	32768	64.27%	64.57%	61.18%	62.56%	63.46%	61.91%	64.18%	64.73%	63.56%	63.22%	63.96%	61.99%
period=			4095			>32768			>32768			>32768	
num_diff_pairs=			4095			8550			7609			7683	
last_pair_pos=			4095			32767			32749			32765	

circuit name=		s1196											
n	2^n	L(14)			G(5, 14, 15)			G2(5, 17[14])			G3(23[14])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	2.42%	3.46%	2.42%	3.22%	6.36%	0.73%	3.22%	6.36%	0.73%	3.22%	6.36%	0.73%
1	2	4.27%	5.56%	4.27%	5.23%	9.50%	3.46%	5.23%	9.50%	3.46%	5.23%	9.50%	3.46%
2	4	5.40%	6.60%	5.40%	7.01%	11.51%	6.20%	6.60%	10.95%	5.64%	7.17%	11.35%	5.88%
3	8	9.74%	10.87%	9.74%	14.49%	17.39%	14.17%	8.37%	12.80%	7.57%	10.31%	13.85%	9.18%
4	16	25.68%	26.17%	25.68%	24.72%	26.65%	24.56%	12.72%	16.02%	12.16%	15.22%	19.08%	14.98%
5	32	29.47%	29.95%	29.47%	35.02%	35.27%	34.94%	21.34%	22.38%	21.18%	40.42%	42.51%	40.34%
6	64	41.87%	42.03%	41.87%	42.35%	42.51%	42.27%	50.32%	50.81%	50.24%	52.42%	54.35%	52.34%
7	128	54.91%	54.91%	54.91%	55.40%	55.56%	55.31%	59.66%	60.23%	59.66%	59.66%	61.43%	59.58%
8	256	65.46%	65.46%	65.46%	71.01%	71.10%	70.93%	67.47%	67.96%	67.47%	70.45%	72.22%	70.45%
9	512	78.02%	78.02%	78.02%	76.49%	76.57%	76.41%	73.75%	74.24%	73.75%	80.03%	80.03%	80.03%
10	1024	83.25%	83.25%	83.25%	84.46%	84.62%	84.46%	81.32%	81.72%	81.32%	86.07%	86.07%	86.07%
11	2048	88.00%	88.00%	88.00%	88.33%	88.49%	88.33%	86.88%	87.04%	86.88%	89.37%	89.37%	89.37%
12	4096	92.27%	92.27%	92.27%	92.03%	92.11%	92.03%	91.87%	91.87%	91.87%	91.14%	91.14%	91.14%
13	8192	94.44%	94.44%	94.44%	96.30%	96.38%	96.30%	93.32%	93.32%	93.32%	93.32%	93.32%	93.32%
14	16384	95.33%	95.33%	95.33%	97.83%	97.91%	97.83%	94.85%	94.85%	94.85%	94.93%	94.93%	94.93%
15	32768	95.33%	95.33%	95.33%	98.39%	98.39%	98.39%	95.17%	95.17%	95.17%	95.65%	95.65%	95.65%
period=			16383			>32768			19530			>32768	
num_diff_pairs=			16383			31366			13978			19263	
last_pair_pos=			16383			32767			19520			32766	

circuit name= s1238

n	2^n patterns	L(14)			G(5, 14, 15)			G2(5, 17[14])			G3(23[14])		
		FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	2.21%	3.10%	2.21%	3.03%	5.76%	0.66%	3.03%	5.76%	0.66%	3.03%	5.76%	0.66%
1	2	3.54%	4.65%	3.54%	4.95%	8.71%	3.25%	4.95%	8.71%	3.25%	4.95%	8.71%	3.25%
2	4	4.80%	5.83%	4.80%	6.35%	10.11%	5.31%	6.20%	9.89%	5.09%	6.20%	9.82%	4.87%
3	8	8.71%	9.67%	8.71%	11.81%	14.24%	11.44%	7.75%	11.51%	6.79%	9.37%	12.47%	8.27%
4	16	23.10%	23.62%	23.10%	21.62%	23.32%	21.48%	11.37%	14.39%	10.78%	13.73%	17.20%	13.51%
5	32	26.49%	27.01%	26.49%	30.78%	31.00%	30.70%	19.19%	20.07%	18.97%	36.90%	38.82%	36.83%
6	64	38.08%	38.30%	38.08%	37.79%	37.93%	37.71%	46.13%	46.57%	46.05%	47.75%	49.59%	47.68%
7	128	50.26%	50.33%	50.26%	50.63%	50.78%	50.55%	54.39%	54.91%	54.39%	54.91%	56.61%	54.83%
8	256	60.44%	60.52%	60.44%	65.61%	65.68%	65.54%	62.36%	62.80%	62.36%	65.46%	67.16%	65.46%
9	512	73.58%	73.58%	73.58%	71.07%	71.14%	71.00%	69.08%	69.52%	69.08%	74.76%	74.76%	74.76%
10	1024	78.52%	78.52%	78.52%	79.56%	79.71%	79.56%	77.12%	77.49%	77.12%	80.44%	80.44%	80.44%
11	2048	83.32%	83.32%	83.32%	83.62%	83.76%	83.62%	82.07%	82.21%	82.07%	84.21%	84.21%	84.21%
12	4096	87.45%	87.45%	87.45%	87.09%	87.16%	87.09%	87.23%	87.23%	87.23%	85.83%	85.83%	85.83%
13	8192	89.30%	89.30%	89.30%	91.00%	91.07%	91.00%	88.56%	88.56%	88.56%	88.34%	88.34%	88.34%
14	16384	90.11%	90.11%	90.11%	92.47%	92.55%	92.47%	89.96%	89.96%	89.96%	89.89%	89.89%	89.89%
15	32768	90.11%	90.11%	90.11%	93.06%	93.06%	93.06%	90.26%	90.26%	90.26%	90.70%	90.70%	90.70%
period=			16383			>32768			19530			>32768	
num_diff_pairs=			16383			31366			13978			19263	
last_pair_pos=			16383			32767			19520			32766	

circuit name= s15850

n	2^n	L(14)			G(5, 14, 15)			G2(5, 17[14])			G3(23[14])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
1	2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
2	4	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
3	8	4.24%	3.47%	1.81%	4.01%	3.43%	1.47%	4.01%	3.43%	1.47%	4.09%	3.48%	1.52%
4	16	6.43%	5.83%	2.93%	5.88%	5.42%	3.14%	7.01%	6.02%	3.91%	7.30%	6.36%	3.98%
5	32	7.95%	7.28%	4.18%	9.48%	9.51%	7.06%	11.32%	9.04%	5.66%	10.08%	8.03%	5.59%
6	64	11.90%	9.95%	7.56%	10.60%	11.16%	8.98%	12.90%	11.64%	8.44%	10.81%	9.29%	6.81%
7	128	15.39%	13.48%	10.47%	12.17%	12.84%	11.04%	15.13%	14.33%	9.98%	11.51%	10.26%	8.16%
8	256	17.24%	16.79%	13.52%	12.40%	13.04%	11.31%	16.20%	15.20%	11.61%	12.31%	10.88%	9.20%
9	512	18.83%	19.86%	14.76%	20.73%	21.65%	16.90%	20.03%	20.17%	14.72%	16.80%	16.73%	11.75%
10	1024	20.84%	21.27%	15.24%	22.24%	23.26%	18.52%	22.41%	22.81%	16.90%	19.52%	19.32%	14.93%
11	2048	22.66%	22.87%	16.14%	24.76%	25.63%	19.26%	24.96%	25.46%	19.19%	20.01%	19.76%	15.42%
12	4096	32.58%	32.69%	23.17%	33.48%	34.62%	23.86%	33.53%	34.12%	24.43%	28.06%	29.35%	20.24%
13	8192	35.83%	36.22%	25.86%	37.03%	38.19%	26.30%	36.99%	37.53%	27.12%	35.72%	36.38%	26.11%
14	16384	40.95%	41.28%	30.89%	38.86%	40.06%	28.18%	39.36%	39.86%	29.08%	37.77%	38.41%	28.15%
15	32768	42.69%	43.64%	32.49%	40.35%	41.29%	30.06%	41.48%	42.59%	31.13%	40.59%	40.54%	29.88%
period=			16383			>32768			19530			>32768	
num_diff_pairs=			16383			31366			13978			19263	
last_pair_pos=			16383			32767			19520			32766	

circuit name= s953

n	2^n patterns	L(16) FF->0	L(16) FF->1	L(16) FF->UD	G(5, 16, 17) FF->0	G(5, 16, 17) FF->1	G(5, 16, 17) FF->UD	G2(5, 19[16]) FF->0	G2(5, 19[16]) FF->1	G2(5, 19[16]) FF->UD	G3(23[16]) FF->0	G3(23[16]) FF->1	G3(23[16]) FF->UD
0	1	2.13%	2.13%	0.00%	2.13%	2.13%	0.00%	2.13%	2.13%	0.00%	2.13%	2.13%	0.00%
1	2	13.62%	13.72%	2.32%	13.81%	14.83%	2.69%	13.81%	14.83%	2.69%	13.81%	14.83%	2.69%
2	4	25.49%	27.90%	2.87%	25.02%	27.43%	2.87%	25.02%	27.43%	2.87%	25.02%	27.43%	2.87%
3	8	35.68%	36.70%	4.36%	33.36%	34.57%	3.71%	33.27%	35.03%	4.73%	35.03%	36.15%	4.36%
4	16	40.22%	41.15%	8.06%	37.91%	39.02%	7.60%	40.13%	41.98%	7.79%	40.13%	41.15%	7.88%
5	32	43.56%	44.76%	8.16%	42.73%	44.12%	7.69%	41.15%	43.00%	7.97%	57.18%	57.09%	8.16%
6	64	53.11%	53.85%	8.34%	75.44%	75.53%	8.34%	67.56%	68.21%	8.25%	61.45%	61.72%	8.25%
7	128	70.81%	70.81%	8.34%	81.28%	81.28%	8.34%	79.80%	79.98%	8.25%	64.78%	65.06%	8.25%
8	256	82.58%	82.58%	8.34%	86.75%	86.75%	8.34%	81.74%	81.84%	8.34%	80.91%	81.00%	8.25%
9	512	87.03%	87.03%	8.34%	89.81%	89.81%	8.34%	88.60%	88.69%	8.34%	86.10%	86.10%	8.34%
10	1024	92.31%	92.31%	8.34%	91.20%	91.20%	8.34%	96.94%	96.94%	8.34%	93.33%	93.33%	8.34%
11	2048	96.39%	96.39%	8.34%	97.59%	97.59%	8.34%	98.33%	98.33%	8.34%	93.42%	93.42%	8.34%
12	4096	98.24%	98.24%	8.34%	98.89%	98.89%	8.34%	98.70%	98.70%	8.34%	93.42%	93.42%	8.34%
13	8192	98.89%	98.89%	8.34%	99.07%	99.07%	8.34%	98.70%	98.70%	8.34%	93.42%	93.42%	8.34%
14	16384	98.89%	98.89%	8.34%	99.07%	99.07%	8.34%	99.07%	99.07%	8.34%	93.42%	93.42%	8.34%
15	32768	98.89%	98.89%	8.34%	99.07%	99.07%	8.34%	99.07%	99.07%	8.34%	93.42%	93.42%	8.34%
period=			>32768			>32768			>32768			>32768	
num_diff_pairs=			32767			32507			28217			1099	
last_pair_pos=			32767			32767			32767			1099	

circuit name= s1423

n	2^n	L(17)			G(5, 17, 18)			G2(5, 19[17])			G3(23[17])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	0.86%	0.66%	0.13%	0.86%	0.66%	0.13%	0.86%	0.66%	0.13%	0.86%	0.66%	0.13%
1	2	1.65%	1.12%	0.86%	1.72%	1.25%	0.92%	1.72%	1.25%	0.92%	1.72%	1.25%	0.92%
2	4	1.65%	1.19%	0.92%	6.87%	7.06%	1.19%	6.87%	7.06%	1.19%	6.87%	7.06%	1.19%
3	8	1.65%	1.19%	0.92%	6.87%	7.13%	1.25%	7.00%	7.13%	4.95%	7.00%	7.20%	6.21%
4	16	5.08%	5.22%	4.49%	7.86%	8.45%	5.55%	7.66%	8.25%	7.33%	8.65%	9.18%	8.25%
5	32	5.08%	5.35%	4.49%	10.83%	11.82%	10.23%	9.70%	10.63%	9.51%	12.08%	12.81%	11.82%
6	64	5.08%	5.35%	4.49%	13.33%	14.26%	12.61%	17.36%	18.15%	16.96%	22.05%	22.71%	21.45%
7	128	8.98%	9.64%	8.65%	14.85%	15.84%	14.52%	24.69%	25.35%	24.16%	25.81%	26.34%	25.08%
8	256	21.32%	21.91%	20.59%	27.79%	28.58%	26.87%	26.07%	26.80%	25.41%	30.50%	31.09%	29.64%
9	512	26.54%	27.20%	25.81%	35.38%	36.11%	34.39%	34.52%	35.31%	33.73%	35.51%	36.24%	34.59%
10	1024	33.33%	34.06%	32.54%	38.81%	39.54%	37.82%	40.66%	41.32%	39.54%	39.27%	40.07%	38.22%
11	2048	37.82%	38.28%	36.70%	51.29%	51.42%	49.51%	41.98%	42.71%	40.86%	42.44%	43.10%	41.32%
12	4096	38.28%	38.61%	36.90%	53.33%	53.33%	51.42%	44.29%	45.02%	43.17%	44.69%	45.35%	43.30%
13	8192	38.61%	38.94%	37.23%	54.79%	54.85%	52.87%	48.65%	49.37%	47.46%	44.75%	45.41%	43.37%
14	16384	38.61%	38.94%	37.23%	55.38%	55.45%	53.40%	50.96%	51.49%	49.57%	44.75%	45.41%	43.37%
15	32768	38.61%	38.94%	37.23%	57.62%	57.76%	55.51%	53.47%	53.99%	52.01%	44.75%	45.41%	43.37%
period=			>32768			>32768			>32768			>32768	
num_diff_pairs=			32767			32664			30226			4219	
last_pair_pos=			32767			32767			32767			4336	

circuit name= s820

n	2^n	L(18)			G(5, 18, 19)			G2(5, 23[18])			G3(25[18])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	5.77%	3.88%	1.06%	5.77%	3.88%	1.06%	5.77%	3.88%	1.06%	5.77%	3.88%	1.06%
1	2	6.35%	8.82%	6.35%	9.65%	6.35%	1.06%	9.65%	6.35%	1.06%	9.65%	6.35%	1.06%
2	4	11.06%	13.29%	11.06%	10.82%	11.65%	7.18%	10.59%	11.06%	6.59%	10.59%	11.06%	6.59%
3	8	12.59%	14.82%	12.59%	14.47%	17.18%	13.29%	13.06%	14.12%	10.24%	14.35%	16.94%	13.18%
4	16	13.18%	15.18%	13.06%	15.41%	18.47%	14.59%	14.24%	16.00%	12.24%	19.65%	21.29%	18.59%
5	32	13.65%	15.65%	13.53%	29.29%	30.24%	28.47%	30.82%	31.41%	29.88%	20.47%	22.12%	19.53%
6	64	14.24%	16.24%	14.12%	30.24%	31.77%	30.00%	31.65%	33.06%	31.53%	26.82%	27.77%	25.88%
7	128	30.71%	30.94%	30.59%	32.12%	33.53%	32.00%	34.35%	35.77%	34.24%	28.00%	29.53%	27.88%
8	256	35.41%	35.65%	35.29%	35.06%	36.47%	34.94%	38.24%	39.65%	38.12%	28.59%	30.12%	28.47%
9	512	36.71%	36.94%	36.59%	35.18%	36.59%	35.06%	39.88%	41.29%	39.77%	34.00%	35.41%	33.88%
10	1024	37.06%	37.29%	36.94%	42.82%	44.12%	42.71%	41.88%	43.29%	41.77%	39.88%	41.29%	39.77%
11	2048	37.88%	38.12%	37.77%	43.41%	44.71%	43.29%	42.24%	43.65%	42.12%	40.35%	41.77%	40.24%
12	4096	40.24%	40.47%	40.12%	46.12%	47.41%	46.00%	43.41%	44.82%	43.29%	41.53%	42.94%	41.41%
13	8192	40.24%	40.47%	40.12%	47.29%	48.59%	47.18%	45.06%	46.47%	44.94%	43.18%	44.59%	43.06%
14	16384	40.24%	40.47%	40.12%	48.59%	49.88%	48.47%	45.29%	46.71%	45.18%	45.18%	46.59%	45.06%
15	32768	40.24%	40.47%	40.12%	48.94%	50.24%	48.82%	46.47%	47.88%	46.35%	46.59%	48.00%	46.47%
period=			>32768			>32768			>32768			>32768	
num_diff_pairs=			32767			32704			31581			31604	
last_pair_pos=			32767			32767			32767			32767	

circuit name= s832

n	2^n patterns	L(18) FF->0	L(18) FF->1	L(18) FF->UD	G(5, 18, 19) FF->0	G(5, 18, 19) FF->1	G(5, 18, 19) FF->UD	G2(5, 23[18]) FF->0	G2(5, 23[18]) FF->1	G2(5, 23[18]) FF->UD	G3(25[18]) FF->0	G3(25[18]) FF->1	G3(25[18]) FF->UD
0	1	5.63%	3.79%	1.03%	5.63%	3.79%	1.03%	5.63%	3.79%	1.03%	5.63%	3.79%	1.03%
1	2	6.21%	8.62%	6.21%	9.43%	6.21%	1.03%	9.43%	6.21%	1.03%	9.43%	6.21%	1.03%
2	4	10.81%	12.99%	10.81%	10.58%	11.38%	7.01%	10.35%	10.81%	6.44%	10.35%	10.81%	6.44%
3	8	12.30%	14.48%	12.30%	14.02%	16.67%	12.87%	12.76%	13.79%	10.00%	14.02%	16.55%	12.87%
4	16	12.87%	14.83%	12.76%	14.94%	17.93%	14.14%	13.91%	15.63%	11.95%	19.20%	20.81%	18.16%
5	32	13.33%	15.29%	13.22%	28.39%	29.31%	27.59%	30.12%	30.69%	29.20%	19.89%	21.49%	18.97%
6	64	13.91%	15.86%	13.79%	29.43%	30.92%	29.20%	30.92%	32.30%	30.81%	26.09%	27.01%	25.17%
7	128	30.00%	30.23%	29.89%	31.26%	32.64%	31.15%	33.45%	34.83%	33.33%	27.24%	28.74%	27.13%
8	256	34.60%	34.83%	34.48%	34.14%	35.52%	34.02%	37.47%	38.85%	37.36%	27.82%	29.31%	27.70%
9	512	35.86%	36.09%	35.75%	34.25%	35.63%	34.14%	39.08%	40.46%	38.97%	33.10%	34.48%	32.99%
10	1024	36.21%	36.44%	36.09%	41.72%	42.99%	41.61%	41.03%	42.41%	40.92%	38.97%	40.35%	38.85%
11	2048	37.01%	37.24%	36.90%	42.30%	43.56%	42.18%	41.38%	42.76%	41.26%	39.43%	40.81%	39.31%
12	4096	39.31%	39.54%	39.20%	44.94%	46.21%	44.83%	42.53%	43.91%	42.41%	40.58%	41.95%	40.46%
13	8192	39.31%	39.54%	39.20%	46.32%	47.59%	46.21%	44.14%	45.52%	44.02%	42.30%	43.68%	42.18%
14	16384	39.31%	39.54%	39.20%	47.59%	48.85%	47.47%	44.37%	45.75%	44.25%	44.37%	45.75%	44.25%
15	32768	39.31%	39.54%	39.20%	47.82%	49.08%	47.70%	45.52%	46.90%	45.40%	45.63%	47.01%	45.52%
period=			>32768			>32768			>32768			>32768	
num_diff_pairs=			32767			32704			31581			31604	
last_pair_pos=			32767			32767			32767			32767	

circuit name=		s420											
n	2^n	L(19)			G(5, 19, 20)			G2(5, 21[19])			G3(25[19])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	0.22%	0.22%	0.22%	0.22%	0.22%	0.22%	0.22%	0.22%	0.22%	0.22%	0.22%	0.22%
1	2	4.84%	1.32%	0.22%	1.98%	1.98%	1.32%	1.98%	1.98%	1.32%	1.98%	1.98%	1.32%
2	4	5.71%	3.52%	1.54%	2.42%	2.42%	1.76%	3.52%	3.08%	2.20%	3.52%	3.08%	2.20%
3	8	6.81%	5.93%	2.86%	4.84%	4.84%	3.30%	5.50%	5.06%	3.30%	6.81%	8.13%	3.74%
4	16	13.63%	22.86%	4.18%	8.79%	9.01%	5.28%	14.73%	12.31%	3.74%	12.09%	15.60%	5.28%
5	32	16.70%	26.15%	4.62%	9.89%	12.53%	5.28%	17.58%	13.19%	5.28%	13.63%	25.28%	5.28%
6	64	17.80%	29.89%	5.06%	22.64%	17.80%	5.28%	30.99%	24.84%	5.28%	18.90%	25.50%	5.28%
7	128	32.31%	37.58%	5.06%	23.74%	21.76%	5.28%	34.73%	34.51%	5.28%	33.41%	28.35%	5.28%
8	256	36.04%	48.57%	5.06%	32.31%	30.33%	5.50%	42.64%	36.92%	5.28%	41.98%	41.32%	5.28%
9	512	48.57%	50.99%	5.06%	42.64%	40.44%	5.50%	44.62%	45.28%	5.28%	43.30%	43.08%	5.28%
10	1024	54.29%	60.88%	5.06%	47.03%	43.52%	5.50%	46.37%	48.35%	5.28%	45.06%	45.93%	5.28%
11	2048	56.70%	66.15%	5.06%	51.43%	48.13%	5.50%	53.63%	53.63%	5.28%	61.98%	63.08%	5.28%
12	4096	60.22%	68.57%	5.50%	60.22%	53.85%	5.71%	60.66%	58.02%	5.28%	66.81%	69.23%	5.28%
13	8192	72.09%	72.31%	5.50%	72.09%	59.34%	5.71%	70.11%	60.88%	5.28%	69.89%	71.65%	5.28%
14	16384	78.90%	79.12%	5.50%	73.85%	68.13%	5.71%	73.41%	66.59%	5.28%	73.41%	78.02%	5.28%
15	32768	84.84%	81.76%	5.50%	76.92%	76.48%	5.71%	82.86%	76.70%	5.71%	84.40%	83.52%	5.28%
period=			>32768			>32768			>32768			>32768	
num_diff_pairs=			32767			32745			32098			32203	
last_pair_pos=			32767			32767			32767			32767	

circuit name=		s510											
n	2^n	L(19)			G(5, 19, 20)			G2(5, 21[19])			G3(25[19])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	7.27%	7.45%	0.00%	7.27%	7.45%	0.00%	7.27%	7.45%	0.00%	7.27%	7.45%	0.00%
1	2	18.26%	20.39%	0.00%	18.26%	20.39%	0.00%	18.26%	20.39%	0.00%	18.26%	20.39%	0.00%
2	4	25.89%	26.06%	0.00%	22.34%	24.29%	0.00%	22.34%	24.29%	0.00%	22.34%	24.29%	0.00%
3	8	51.42%	30.85%	0.00%	37.59%	25.89%	0.00%	48.40%	25.89%	0.00%	48.40%	25.89%	0.00%
4	16	56.38%	45.21%	0.00%	42.38%	36.17%	0.00%	56.03%	40.07%	0.00%	56.56%	38.83%	0.00%
5	32	59.57%	72.52%	0.00%	43.62%	54.61%	0.00%	70.57%	78.01%	0.00%	72.87%	78.55%	0.00%
6	64	79.26%	89.01%	0.00%	57.98%	79.97%	0.00%	89.54%	92.20%	0.00%	88.12%	91.14%	0.00%
7	128	96.10%	95.75%	0.00%	92.55%	97.16%	0.00%	96.63%	96.10%	0.00%	97.52%	95.92%	0.00%
8	256	98.58%	98.76%	0.00%	98.58%	98.94%	0.00%	98.76%	98.58%	0.00%	98.40%	98.40%	0.00%
9	512	99.82%	99.82%	0.00%	99.82%	99.82%	0.00%	100.00%	100.00%	0.00%	100.00%	98.94%	0.00%
10	1024	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%
11	2048	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%
12	4096	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%
13	8192	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%
14	16384	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%
15	32768	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%
period=			>32768			>32768			>32768			>32768	
num_diff_pairs=			32767			32745			32098			32203	
last_pair_pos=			32767			32767			32767			32767	

circuit name=		s9234											
n	2^n	L(19)			G(5, 19, 20)			G2(5, 21[19])			G3(25[19])		
	patterns	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD	FF->0	FF->1	FF->UD
0	1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
1	2	1.30%	1.91%	0.87%	1.29%	1.92%	0.48%	1.29%	1.92%	0.48%	1.29%	1.92%	0.48%
2	4	4.07%	3.96%	1.10%	2.12%	3.68%	1.17%	2.19%	3.77%	1.21%	2.19%	3.77%	1.21%
3	8	5.75%	6.01%	3.34%	5.53%	6.02%	1.57%	2.90%	4.75%	1.52%	2.87%	4.79%	1.56%
4	16	8.60%	7.09%	3.64%	8.79%	7.33%	3.42%	6.94%	6.35%	2.38%	6.77%	6.40%	2.48%
5	32	9.89%	8.21%	4.98%	9.01%	7.57%	3.97%	9.15%	8.19%	5.02%	9.23%	8.29%	5.13%
6	64	10.15%	8.40%	5.18%	9.95%	8.52%	5.21%	9.25%	8.29%	5.13%	9.30%	8.36%	5.17%
7	128	10.34%	8.95%	5.28%	10.00%	8.81%	5.24%	12.85%	8.50%	5.13%	12.18%	8.88%	5.26%
8	256	13.12%	8.98%	5.31%	10.47%	12.30%	5.31%	13.04%	8.79%	5.31%	12.43%	8.91%	5.31%
9	512	13.90%	8.98%	5.31%	10.76%	12.43%	5.31%	13.18%	8.91%	5.31%	12.78%	9.07%	5.31%
10	1024	14.13%	9.07%	5.31%	13.44%	12.60%	5.31%	13.61%	9.07%	5.31%	12.83%	9.31%	5.31%
11	2048	14.90%	9.28%	5.41%	13.99%	13.25%	5.41%	14.05%	9.27%	5.41%	14.54%	9.50%	5.41%
12	4096	17.70%	10.21%	5.50%	16.28%	16.10%	6.11%	17.57%	10.19%	5.41%	14.77%	13.99%	5.41%
13	8192	17.96%	12.14%	5.50%	16.95%	18.28%	6.45%	18.18%	15.46%	6.14%	15.45%	18.05%	6.14%
14	16384	18.38%	12.78%	6.22%	17.34%	20.33%	7.03%	18.61%	15.65%	6.34%	17.67%	19.09%	6.68%
15	32768	18.96%	13.28%	6.24%	17.73%	21.47%	7.05%	19.49%	20.11%	7.05%	18.48%	19.76%	7.05%
period=			>32768			>32768			>32768			>32768	
num_diff_pairs=			32767			32745			32098			32203	
last_pair_pos=			32767			32767			32767			32767	

Appendix 3: Software Documentation

In this research, we conduct experiments on four types of test pattern generators: the LFSR, the conventional Geffe generator, modified Geffe 2 and Geffe 3. They are implemented using the C++ programming language to facilitate an easy pattern generation for the fault simulation. This appendix provides the implementation details of these generators.

1. LFSR Implementation

The implementation of LFSR is given in *MyLFSR.cc*. This program allows users to specify the characteristic polynomial for a LFSR. It also provides some degree of freedom for users to initialize the length of test vectors and the size of the test pattern file. Upon execution, the command prompt guides users to input all required parameters. According to the user instructions, *MyLFSR.cc* produces a text file, which contains a list of test patterns for the Hope simulator.

The following is the source code for the LFSR:

```
//*******************************************
// MyLFSR.cc
//
//
// Author: Diane Qi
// Date: July 13, 2004
//*******************************************


#include <iostream>
#include <string>
#include <fstream.h>

using namespace std;

/*********************************************
 * ---hasEXOR(int i, int poly[], int ary_size)
 * a function which checks whether state Si has
 * an Exor gate preceded. The decision is made
 * based on a given polynormial function poly[]
 * with size of ary_size
```

```
 *
 *@return:   integer 0 if no Exor before Si
 *           integer 1 if an Exor preceed Si
 ***********************************************/

int hasEXOR(int i, int poly[], int ary_size){
    int n;
    int ans=0; //initial as Si state is not preceded by Exor gate

    for(n=1; n< ary_size-1; n++){
            if(i==poly[n]){
                    ans=1; //found such state
                    break;
            }
    }

    return ans;

}


/*****************************************

 *---printArray(int ary[], int ary_size)
 * a pretty printer for outputting a list
 * of integers
 *
 * @return: void
 ****************************************/
void printArray(int ary[], int ary_size){
    int n;
    for (n=0;n<ary_size ;n++) {
                    cout <<ary[n];
                    cout <<"  ";
            }
            cout<< endl;

}


main(int argc, char** argv)
{
    /*variable declaration*/

    int M;          //number of vectors you need to generate
    int LENGTH;     // pattern length(cut length)
    int N1;         //degree of LFSR1
    int K1;         //weight of polynomial for LFSR1

    char* output;     //name of the output file
    char* output_mode; //0-->select first n bits
                       //1-->select n equally distributed bits
    output=argv[1]; //obtain the outfile name
    output_mode=argv[2];
```

```
/*obtain variables for Geffe1 base setting*/

cout << "*======Set up a LFSR machine for Experiment#1======";
cout << endl;
cout << "*Please enter the polynomial function for LFSR1";
cout << "*(starting from the highest power to 0):";

cout << endl;
cout << "*";
cin>>N1;
K1=0;
int poly1[N1+1]; //polynomial of LFSR1
                 //the maximum size of poly function is N1+1
poly1[K1]=N1;
do{
        K1++;
        cout << "*";
        cin>>poly1[K1];
}while(poly1[K1] != 0);
K1++;



cout<< "*Please enter the pattern length:";
cin>>LENGTH;

cout<< "*Please enter the number of patterns you want to collect:";
cin>>M;

/*compute the which cell is used for outputting bits*/
/*equally distributed bits is the default case*/
int z; int active_bits[LENGTH];
for(z=0; z<LENGTH; z++){
        active_bits[z]=z*N1/LENGTH;
}
/*if user want to collect test pattern from the first n cells*/
if(output_mode[0]=='0'){
        for(z=0; z<LENGTH; z++){
                active_bits[z]=z;
        }
}

/*summary of the LFSR setting*/
ofstream outfile(output);
outfile << "*LFSR1:";
cout<<"*LFSR1:";
int n;
for (n=0;n<K1 ;n++) {
                outfile <<poly1[n];
                cout <<poly1[n];
                outfile <<" ";
                cout <<" ";
        }
        outfile<< endl;
        cout << endl;
```

```
outfile << "*take bits from position:";
cout <<"*take bits from position:";
for (n=0;n<LENGTH ;n++) {
                outfile <<active_bits[n];
                cout<<active_bits[n];
                outfile <<"  ";
                cout <<"  ";
        }
        outfile<< endl;
        cout<<endl;

outfile << "*";
outfile << "pattern_length:" << LENGTH; //debug
outfile << endl;
outfile << "*";
outfile << "num_patterns:" << M; //debug
outfile << endl;

/*Generate patterns*/
bool a[M][N1];   //state table mapped to LFSR1

int f;
int i,j,l,p;
int q,s;

/*initial state of LFSR1: 00...001*/
for (l=0;l<N1-1;l++)
{
        a[0][l]=0;

}
a[0][N1-1]=1;

/*construct a state table for LFSR1 with M time slices*/
for (i=1;i<M;i++)
{
        a[i][0]=a[i-1][N1-1];
        for (j=1;j<N1;j++)
        {

                int has_Exor=hasEXOR(j, poly1, K1);
                if(has_Exor==1){
                        a[i][j]=a[i-1][N1-1]^a[i-1][j-1];
                }
                else{
                        a[i][j]=a[i-1][j-1];
                }
        }
}



/*format output
*1:0101110
*2:1110011
*3:....
```

```
*.....
*/

for (q=0;q<M;q++)
{
        outfile<<q+1<<":";
        for (s=0;s<LENGTH;s++){
                int pos=active_bits[s];
                        outfile<<a[q][pos];
                }

        outfile<<endl;
}

} //end of main()
```

2. Conventional Geffe Generator Implementation

The conventional Geffe generator is implemented by *MyGeffe.cc*. This program allows users to specify the characteristic polynomial for the three LFSRs in a Geffe machine. Similar to *MyLFSR.cc*, it also provides users some degree of freedom to initialize the length of test vectors and the size of the test pattern file. According to the user instructions, *MyGeffe.cc* produces a text file which contains a list of test patterns. This file is formatted correctly in order to plug-in to the Hope simulator directly.

MyGeffe.cc is described in the following:

```
//*********************************************
// MyGeffe.cc
//
// regular Geffe generator
// LFSR1 and LFSR2 are the 2 source LFSR
// LFSR3 acts as a selector for Mux2:1.
//
// Author: Diane Qi
// Date: July 13, 2004
//*********************************************


#include <iostream>
#include <string>
#include <fstream.h>

using namespace std;

/*********************************************
```

```
 * ---hasEXOR(int i, int poly[], int ary_size)
 * a function which checks whether state Si has
 * an Exor gate preceded. The decision is made
 * based on a given polynormial function poly[]
 * with size of ary_size
 *
 *@return:   integer 0 if no Exor before Si
 *           integer 1 if an Exor preceed Si
 ***********************************************/

int hasEXOR(int i, int poly[], int ary_size){
    int n;
    int ans=0; //initial as Si state is not preceded by Exor gate

    for(n=1; n< ary_size-1; n++){
            if(i==poly[n]){
                    ans=1; //found such state
                    break;
            }
    }

    return ans;

}


/*****************************************

 *---printArray(int ary[], int ary_size)
 * a pretty printer for outputing a list
 * of integers
 *
 * @return: void
 ****************************************/
void printArray(int ary[], int ary_size){
    int n;
    for (n=0;n<ary_size ;n++) {
                    cout <<ary[n];
                    cout <<"  ";
            }
            cout<< endl;

}


main(int argc, char** argv)
{
    /*variable declaration*/

    int M;          //number of vectors you need to generate
    int LENGTH;     // pattern length(cut length)
    int N1;         //degree of LFSR1
    int N2;         //degree of LFSR2
    int N3;         //degree of LFSR3
    int K1;         //weight of polynomial for LFSR1
    int K2;         //weight of polynomial for LFSR2
```

```
int K3;             //weight of polynomial for LFSR3

char* output;       //name of the output file

output=argv[1]; //obtain the outfile name

/*obtain variables for Geffe1 base setting*/

cout << "*======Set up a Geffe machine for Experiment#2======";
cout << endl;
cout << "*Please enter the polynormial function for LFSR1";
cout << "*(starting from the highest power to 0):";

cout << endl;
cout << "*";
cin>>N1;
K1=0;
int poly1[N1+1]; //polynomial of LFSR1
                 //the maximum size of poly function is N1+1
poly1[K1]=N1;
do{
        K1++;
        cout << "*";
        cin>>poly1[K1];
}while(poly1[K1] != 0);
K1++;

cout << "*Please enter the polynormial function for LFSR2";
cout <<  "(starting from the highest power to 0):";
cout << endl;
cout << "*";
cin >> N2;
K2=0;
int poly2[N2+1];//polynomial of LFSR2
                 //the maximum size of poly function is N1+1
poly2[K2]=N2;
do{
        K2++;
        cout << "*";
        cin  >> poly2[K2];
}while(poly2[K2] != 0);
K2++;

cout << "*Please enter the polynomial function for LFSR3";
cout <<  "(starting from the highest power to 0):";
cout << endl;
cout << "*";
cin >> N3;
K3=0;
int poly3[N3+1];//polynomial of LFSR2
                 //the maximum size of poly function is N3+1
poly3[K3]=N3;
do{
        K3++;
        cout << "*";
        cin  >> poly3[K3];
```

```
}while(poly3[K3] != 0);
K3++;

//cout << "*LFSR3:";
//printArray(poly3, K3); //debug



cout<< "*Please enter the pattern length:";
cin>>LENGTH;

cout<< "*Please enter the number of patterns you want to collect:";
cin>>M;



/*summary of the Geffe setting*/
ofstream outfile(output);
outfile << "*LFSR1:";
int n;
for (n=0;n<K1 ;n++) {
                    outfile <<poly1[n];
                    outfile <<"  ";
          }
          outfile<< endl;
outfile << "*LFSR2:";
for (n=0;n<K2 ;n++) {
                    outfile <<poly2[n];
                    outfile <<"  ";
          }
          outfile << endl;
outfile << "*Selector LFSR:";
for (n=0;n<K3 ;n++) {
                    outfile <<poly3[n];
                    outfile <<"  ";
          }
          outfile << endl;


outfile << "*";
outfile << "pattern_length:" << LENGTH; //debug
outfile << endl;
outfile << "*";
outfile << "num_patterns:" << M; //debug
outfile << endl;

/*Generate patterns*/

bool a[M][N1];   //state table mapped to LFSR1
bool b[M][N2];   //state table mapped to LFSR2
bool c[M][N3];   //state table mapped to LFSR3 (selector)

int f;
int i,j,l,p;
int q,s;
```

```
/*initial state of LFSR1: 00...001*/
for (l=0;l<N1-1;l++)
{
        a[0][l]=0;

}
a[0][N1-1]=1;

/*construct a state table for LFSR1 with M time slices*/
for (i=1;i<M;i++)
{
        a[i][0]=a[i-1][N1-1];
        for (j=1;j<N1;j++)
        {

                int has_Exor=hasEXOR(j, poly1, K1);
                if(has_Exor==1){
                        a[i][j]=a[i-1][N1-1]^a[i-1][j-1];
                }
                else{
                        a[i][j]=a[i-1][j-1];
                }
        }
}


/*initial state of LFSR2: 00...001*/
for (l=0;l<N2-1;l++)
{
        b[0][l]=0;
}
b[0][N2-1]=1;

/*construct a state table for LFSR2 with M time slices*/
for (i=1;i<M;i++)
{
        b[i][0]=b[i-1][N2-1];
        for (j=1;j<N2;j++)
        {
                int has_Exor=hasEXOR(j, poly2, K2);
                if(has_Exor==1){
                        b[i][j]=b[i-1][N2-1]^b[i-1][j-1];
                }
                else{
                        b[i][j]=b[i-1][j-1];
                }
        }
}

/*initial state of LFSR3: 00...001*/
for (l=0;l<N3-1;l++)
{
        c[0][l]=0;
}
c[0][N3-1]=1;
```

```
/*construct a selector with M time slices*/
for (i=1;i<M;i++)
{
        c[i][0]=c[i-1][N3-1];
        for (j=1;j<N3;j++)
        {
                int has_Exor=hasEXOR(j, poly3, K3);
                if(has_Exor==1){
                        c[i][j]=c[i-1][N3-1]^c[i-1][j-1];
                }
                else{
                        c[i][j]=c[i-1][j-1];
                }
        }
}

/*format output
*1:0101110
*2:1110011
*3:....
*.....
*/
for (q=0;q<M;q++)
{
        outfile<<q+1<<":";
        if (c[q]==0) //if selector bit is 0, choose LFSR1
                for (s=0;s<LENGTH;s++)
                        outfile<<a[q][s];

        else //else choose LFSR2
                for (s=0;s<LENGTH;s++)
                        outfile<<b[q][s];
        outfile<<endl;
}

} //end of main()
```

3. Geffe 2 Implementation

The conventional Geffe generator can be modified to Geffe 2 by applying modification 2 introduced in Chapter 3 section 3.4. Geffe 2 is implemented in a similar manner to the conventional Geffe being implemented. The source code of Geffe 2 is shown in the next page:

```
//*************************************************************
// Geffe2.cc
//
// A modified version of Geffe generator
// This generator. LFSR1 and LFSR2 is combined
// into one single LFSR+. Selector LFSR still
// remains the same.
//
// Author: Diane Qi
// Date: July 13, 2004
//*************************************************************


#include <iostream>
#include <string>
#include <fstream.h>

using namespace std;

/*********************************************
 * ---hasEXOR(int i, int poly[], int ary_size)
 * a function which checks whether state Si has
 * an Exor gate preceded. The decision is made
 * based on a given polynormial function poly[]
 * with size of ary_size
 *
 *@return:      integer 0 if no Exor before Si
 *              integer 1 if an Exor preceed Si
 *********************************************/

int hasEXOR(int i, int poly[], int ary_size){
        int n;
        int ans=0; //initial as Si state is not preceded by Exor gate

        for(n=1; n< ary_size-1; n++){
                if(i==poly[n]){
                        ans=1; //found such state
                        break;
                }
        }

        return ans;

}


/*****************************************

 *---printArray(int ary[], int ary_size)
 * a pretty printer for outputing a list
 * of integers
 *
 * @return: void
 *****************************************/
void printArray(int ary[], int ary_size){
        int n;
```

```
        for (n=0;n<ary_size ;n++) {
                cout <<ary[n];
                cout <<"  ";
        }
        cout<< endl;

}


main(int argc, char** argv)
{
        /*variable declaration*/

        int M;          //number of vectors you need to generate
        int LENGTH;     // pattern length(cut length), or imbed poly degree
        int N1;         //degree of LFSR+
        int N2;         //degree of selector LFSR
        int K1;         //weight of polynomial for LFSR+
        int K2;         //weight of polynomial for selector LFSR

        char* output;   //name of the output file

        output=argv[1]; //obtain the outfile name

        /*obtain variables for Geffe1 base setting*/

        cout << "*======Set up a Geffe machine for Experiment#3======";
        cout << endl;
        cout << "*Please enter the polynormial function for LFSR+";
        cout << "*(starting from the highest power to 0):";

        cout << endl;
        cout << "*";
        cin>>N1;
        K1=0;
        int poly1[N1+1]; //polynomial of LFSR1
                         //the maximum size of poly function is N1+1
        poly1[K1]=N1;
        do{
                K1++;
                cout << "*";
                cin>>poly1[K1];
        }while(poly1[K1] != 0);
        K1++;


        cout << "*Please enter the polynomial function for selector LFSR";
        cout <<  "(starting from the highest power to 0):";
        cout << endl;
        cout << "*";
        cin >> N2;
        K2=0;
        int poly2[N2+1];//polynomial of LFSR2
                        //the maximum size of poly function is N1+1
        poly2[K2]=N2;
        do{
```

```
        K2++;
        cout << "*";
        cin  >> poly2[K2];
}while(poly2[K2] != 0);
K2++;


cout<< "*Please enter the pattern length:";
cin>>LENGTH;

cout<< "*Please enter the number of patterns you want to collect:";
cin>>M;



/*summary of the Geffe setting*/
ofstream outfile(output);
outfile << "*LFSR+:";
int n;
for (n=0;n<K1 ;n++) {
        outfile <<poly1[n];
        outfile <<"  ";
}
outfile<< endl;
outfile << "*selector LFSR:";
for (n=0;n<K2 ;n++) {
        outfile <<poly2[n];
        outfile <<"  ";
}
outfile << endl;


outfile << "*";
outfile << "pattern_length/imbed_length:" << LENGTH; //debug
outfile << endl;
outfile << "*";
outfile << "num_patterns:" << M; //debug
outfile << endl;

/*generate patterns*/

bool a[M][N1];   //state table mapped to LFSR+
bool b[M][N2];   //state table mapped to selector LFSR

int f;
int i,j,l,p;
int q,s;

/*initial state of LFSR+: 00...001*/
for (l=0;l<N1-1;l++)
{
        a[0][l]=0;

}
a[0][N1-1]=1;
```

```
/*initial state of selector LFSR: 00...001*/
for (l=0;l<N2-1;l++)
{
        b[0][l]=0;
}
b[0][N2-1]=1;

/*construct a state table for selector LFSR with M time slices*/
for (i=1;i<M;i++)
{
        b[i][0]=b[i-1][N2-1];
        for (j=1;j<N2;j++)
        {
                int has_Exor=hasEXOR(j, poly2, K2);
                if(has_Exor==1){
                        b[i][j]=b[i-1][N2-1]^b[i-1][j-1];
                }
                else{
                        b[i][j]=b[i-1][j-1];
        }
        }
}

/*construct a state table for LFSR+ with M time slices*/
bool feed_back;
feed_back=1; //first round
for (i=1;i<M;i++)
{
        for (j=0;j<N1;j++)
        {
        if(j < LENGTH && b[i-1][N2-1]==1){ //handle MUX
                feed_back=a[i-1][LENGTH-1];

        }else{
                feed_back=a[i-1][N1-1];
        }
                int has_Exor=hasEXOR(j, poly1, K1);
                if(j==0){
                        a[i][j]=feed_back;
                }else{
                        if(has_Exor==1){
                                a[i][j]=feed_back^a[i-1][j-1];
                        }
                        else {
                                a[i][j]=a[i-1][j-1];
                        }
        }
        }
}


/* debug: output the first 10 for checking*/
cout<<"Selector LFSR";
cout<<endl;
for(q=0; q<10; q++){
        cout<<q+1<<":";
```

```
                for(s=0; s<N2; s++){
                        cout<<b[q][s];

                }
                cout<<endl;
        }

        cout<<"LFSR+";
        cout<<endl;
        for(q=0; q<10; q++){
                cout<<q+1<<":";
                for(s=0; s<N1; s++){
                        cout<<a[q][s];

                }
                cout<<endl;
        }


        /*format output
        *1:0101110
        *2:1110011
        *3:....
        *.....
        */

        for (q=0;q<M;q++)
        {
                outfile<<q+1<<":";
                for (s=0;s<LENGTH;s++)
                        outfile<<a[q][s];

                outfile<<endl;
        }

} //end of main()
```

4. Geffe 3 Implementation

The Geffe generator designed using modification 3 is implemented by *Geffe3.cc*. By entering the characteristic polynomial through the command prompt, the program allows users to specify the $LFSR_4^+$, the only LFSR in Geffe 3. *Geffe3.cc* is shown in the following pages:

```
//*********************************************************
// Geffe3.cc
//
// A modified version of Geffe generator
// This generator only contains a single LFSR and
// a 2-to-1 MUX.
//
// Author: Diane Qi
// Date: July 13, 2004
//*********************************************************


#include <iostream>
#include <string>
#include <fstream.h>

using namespace std;

/*********************************************
 * ---hasEXOR(int i, int poly[], int ary_size)
 * a function which checks whether state Si has
 * an Exor gate preceded. The decision is made
 * based on a given polynormial function poly[]
 * with size of ary_size
 *
 *@return:        integer 0 if no Exor before Si
 *                integer 1 if an Exor preceed Si
 *********************************************/

int hasEXOR(int i, int poly[], int ary_size){
        int n;
        int ans=0; //initial as Si state is not preceded by Exor gate

        for(n=1; n< ary_size-1; n++){
                if(i==poly[n]){
                        ans=1; //found such state
                        break;
                }
        }

        return ans;

}


/*****************************************

 *---printArray(int ary[], int ary_size)
 * a pretty printer for outputing a list
 * of integers
 *
 * @return: void
 *****************************************/
void printArray(int ary[], int ary_size){
        int n;
```

```
        for (n=0;n<ary_size ;n++) {
                cout <<ary[n];
                cout <<"  ";
        }
        cout<< endl;

}


main(int argc, char** argv)
{
        /*variable declaration*/

        int M;          //number of vectors you need to generate
        int LENGTH;     // pattern length(cut length), or imbed poly degree
        int emb_length;
        int N1;         //degree of LFSR1
        int K1;         //weight of polynomial for LFSR1

        char* output;   //name of the output file

        output=argv[1]; //obtain the outfile name

        /*obtain variables for Geffe1 base setting*/

        cout << "*======Set up a Geffe machine for Experiment#4======";
        cout << endl;
        cout << "*Please enter the polynomial function for LFSR+";
        cout << "*(starting from the highest power to 0):";

        cout << endl;
        cout << "*";
        cin>>N1;
        K1=0;
        int poly1[N1+1]; //polynomial of LFSR1
                         //the maximum size of poly function is N1+1
        poly1[K1]=N1;
        do{
                K1++;
                cout << "*";
                cin>>poly1[K1];
        }while(poly1[K1] != 0);
        K1++;

        //cout << "*LFSR1:";

        //printArray(poly1, K1); //debug



        cout<< "*Please enter the embed length:";
        cin>>emb_length;
        cout<< "*Please enter the pattern length:";
        cin>>LENGTH;
```

```
cout<< "*Please enter the number of patterns you want to collect:";
cin>>M;


/*summary of the Geffe setting*/
ofstream outfile(output);
outfile << "*LFSR+:";
int n;
for (n=0;n<K1 ;n++) {
        outfile <<poly1[n];
        outfile <<"  ";
}
outfile<< endl;

outfile << "*";
outfile << "embed_length:" << emb_length; //debug
outfile << endl;

outfile << "*";
outfile << "pattern_length:" << LENGTH; //debug
outfile << endl;

outfile << "*";
outfile << "num_patterns:" << M; //debug
outfile << endl;

/*Generate patterns*/

bool a[M][N1];   //state table mapped to LFSR+

int f;
int i,j,l,p;
int q,s;

/*initial state of LFSR+: 00...001*/
for (l=0;l<N1-1;l++)
{
        a[0][l]=0;

}
a[0][N1-1]=1;


/*construct a state table for LFSR+ with M time slices*/
bool feed_back;
feed_back=1; //first round
for (i=1;i<M;i++)
{
        //a[i][0]=feed_back;
        for (j=0;j<N1;j++)
        {
        if(j < emb_length && a[i-1][LENGTH]==1){ //handle MUX
                feed_back=a[i-1][emb_length-1];
```

```
		}else{
			feed_back=a[i-1][N1-1];
		}
			int has_Exor=hasEXOR(j, poly1, K1);
			if(j==0){
				a[i][j]=feed_back;
			}else{
				if(has_Exor==1){
					a[i][j]=feed_back^a[i-1][j-1];
				}
				else {
					a[i][j]=a[i-1][j-1];
				}
		}
		}
	}


	/*output the first 10 test pattern for checking*/
	cout<<"LFSR+";
	cout<<endl;
	for(q=0; q<10; q++){
		cout<<q+1<<":";
		for(s=0; s<N1; s++){
			cout<<a[q][s];

		}
		cout<<endl;
	}


	/*format output
	*1:0101110
	*2:1110011
	*3:....
	*.....
	*/

	for (q=0;q<M;q++)
	{
		outfile<<q+1<<":";
		for (s=0;s<LENGTH;s++)
			outfile<<a[q][s];

		outfile<<endl;
	}

} //end of main()
```

Appendix 4: Fault Simulation Instruction

The fault simulation experiment is conducted in the DSD (Digital System Design) research group laboratory in the Department of Computer Science of the University of Victoria. This appendix provides guidance to the simulation tools available in the lab.

1. Location of the Resources

The materials used for our simulation include test pattern files, ISCAS'89 benchmark circuit files, and the HOPE simulator.

- Test pattern files are generated by the software introduced in the Appendix 3.
- The ISCAS'89 benchmark set is available on Shannon (UNIX machine) under the following directory:

  shannon/a/csvlsi.020726/benchmarks/ISCAS89/bench/
- The HOPE simulator is also installed on Shannon. It can be evoked in a UNIX console by entering "hope" command. The user menu is illustrated in the next section.

2. User's Guide for HOPE

Together with the ISCAS'89 benchmark circuit files, test patterns generated by various types of generators are fed into HOPE simulation. The HOPE simulator computes the fault coverage accordingly. This result, therefore, is used for the analysis. The following provides the user menu of the HOPE simulator.

NAME: HOPE –a parallel fault simulator for synchronous sequential circuits.

SYNOPSIS:hope [options] circuit_file [> outfile]

OPTIONS: Several options are available for hope. If an option is not specified, the default values are used.

-c fn Options are read from the file "fn".
(default: online command mode)

-D The list of faults which are newly detected by a test pattern is reported in the log file. The option -l should be specified. (default: only the number)

-f fn Faults are defined in the file "fn".

-F fn The good and faulty circuit outputs are reported for each fault in the file fn. In this option, the fault all the faults are injected and simulated in parallel. (default: faulty circuit output is not reported.)

-h f Displays an example fault list format.

-h g Displays the on-line user's guide.

-h n Displays an example netlist format.

-h t Displays an example test pattern file.

-h a Displays the entire on-line manual file.

-l fn A log file named "fn" is created.
(default: no logfile is created)

-N Diagnostic mode
No fault dropping is performed. That is, all the faults are simulated for each test pattern.
(default: faults detected during the fault simulation are dropped from the fault list.)

-r n (Random pattern generation mode)
Test patterns are generated randomly. The fault simulation stops either when all faults are detected or n patterns are applied. (default: -r 224)

-s n Initial seed for the random number generator is set by n.
If n=0, random seed is generated using the day time of the computer. (default: -s 0)

-t fn Test patterns are provided in the file "fn"
(default: random patterns are used)

-u Prints out all undetected faults in a file. The name of this files is <ckt>.ufaults. Note that hope does not update a fault file if one already exists in the run directory. This fault list file can be directly read by atalanta or hope.
(default: no file is created)

-U fn The same as –u, but hope writes undetected faults to the given file name.
(default: no file is created)

-x (Potential detection mode)
Potentially detected faults are dropped as soon as they are detected.
(default: only detected faults are dropped)

-0 All the flip-flops are initially set to logic 0.

-1 All the flip-flops are initially set to logic 1.
(default: All the flip-flops are initially set to unknown (x).)

EXAMPLES:

hope –t s27.test s27.bench

--- simulates the circuit s27.bench using the test patterns in the file "s27.test". The fault simulation stops when all test patterns in the file "s27.test" are simulated or all faults are detected. The summary of the fault simulation is reported to the standard output (CRT terminal).

hope –s 9999 –r 20000 s27.bench > s27.out

--- simulates the circuit s27.bench using 20000 random patterns. The random pattern generator is initialized by 20000. The fault simulation stops when 20000 random patterns are simulated or all faults are detected. The summary of the fault simulation is reported to the file "s27.out".

hope –f s27.fault –t s27.test –l s27.dict –N –D s27.bench

--- reads the fault list from the file "s27.fault" and simulates faults in a diagnosis fault, i.e., no fault dropping is applied. The result of fault simulation is reported in the log file "s27.dict". In the log file, HOPE reports the list of faults detected by each test pattern is listed.

TEST PATTERN FILE:

The line beginning with "*" is a comment line and ignored. Each test pattern begins after a colon (:). For an *n* input circuit, only the *n* bits following ":" are significant, and the remaining bits, if any, are ignored. The following is an example of the test pattern file for s27.

```
****************************************
*file name              :s27.test
*pattern length         :4
*number of patterns     :32768
****************************************
1:0000
2:1001
3:1101
4:1111
5:1110
6:0111
7:1010
8:0101
9:1011
10:1100
...
32766:0100
32767:1011
32768:1100
```

OUTPUTS:

In default mode, no file is created. The summary of the fault simulation is reported to the standard output. If –l option is specified, hope creates a log file. The log file contains more detailed information on the fault simulation. An example of the simulation summary of the circuit s27.bench is shown in the next page. This summary is generated using the default output mode.

```
*************************************************************************
*                                                                       *
*                    Welcome to HOPE (version 2.0)                      *
*                                                                       *
*                       Dong S. Ha (ha@vt.edu)                          *
*                    Web: http://www.ee.vt.edu/ha                       *
*              Virginia Polytechnic Institute & State University        *
*                                                                       *
*************************************************************************
```

****** SUMMARY OF SIMULATION RESULTS ******

1. Circuit structure
   - Name of circuit : s27
   - Number of primary inputs : 4
   - Number of primary outputs : 1
   - Number of flip-flops : 3
   - Number of gates : 10
   - Level of the circuit : 6

2. Simulator input parameters
   - Simulation mode : file (s27.test)

3. Simulation results
   - Number of test patterns applied : 32768
   - Fault coverage : 87.500 %
   - Number of collapsed faults : 32
   - Number of detected faults : 28
   - Number of undetected faults : 4

4. Memory used : 353 Kbytes

5. CPU time
   - Initialization : 0.033 secs
   - Fault simulation : 0.350 secs
   - Total : 0.383 secs