OPTIMAL DESIGN OF VLSI STRUCTURES WITH BUILT-IN SELF TEST
BASED ON REDUCED PSEUDO-EXHAUSTIVE TESTING.

A dissertation Presented to
The Faculty of the College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

by
Tales Cleber Pimenta/
November, 1992
ACKNOWLEDGEMENTS

I wish to express my deep appreciation to my advisor Dr. E. M. Mokari for his guidance, assistance and encouragement, and for acting more like a father than an advisor.

I would like to thank the members of my committee, Dr. Jeffrey Dill, Dr. Herman Hill, Dr. Larry Snyder and Dr. Constantinos Vassiliadis.

I am grateful to Dr. Mitchell for the support during my studies.

I also would like to express my gratitude to my parents for their caring and confidence.

Finally, I dedicate this work to my wife for the support, encouragement and forgiveness in this work.
# Table of Contents

Acknowledgements ................................................................. i

Table of Contents ................................................................. ii

List of Figures ........................................................................ v

List of Tables ........................................................................... ix

List of Abbreviations ............................................................... x

1 Introduction ........................................................................... 1
  1.1 Background ................................................................. 4
  1.2 Design for Testability .................................................... 11
  1.3 Test Generation Methodologies ..................................... 20

2 Linear Feedback Shift Registers ........................................... 23
  2.1 Shift Registers ............................................................ 23
  2.2 Shift Register with Linear Feedback ............................. 26
     2.2.1 Primitive Polynomials ........................................ 28
     2.2.2 Modular LFSR .................................................. 31
  2.3 Linear Codes .............................................................. 33

3 Response Analysis ............................................................... 38
  3.1 Compression Methods .................................................. 39
     3.1.1 ONEs Count ..................................................... 40
3.1.2 Transition Count ................................................. 43
3.1.3 Parity Check ...................................................... 44
3.1.4 Variations of the Basic Methods ............................... 45
3.2 Signature Analysis using LFSR .................................. 47
3.3 Aliasing in LFSR .................................................... 50
3.4 Aliasing Bounds .................................................... 58

4 Pseudo-Exhaustive Testing ........................................... 63
4.1 Partitioning ........................................................ 63
4.2 Logic Sensitization ................................................ 68
4.3 Verification Testing ................................................ 71
  4.3.1 Automatic Generation of Dependence Matrix ............. 73
  4.3.2 Verification Testing based on Rewiring .................... 74
  4.3.3 Verification Testing based on Linear Codes .............. 81
4.4 Segmentation ....................................................... 82

5 Reduced Test Method ................................................ 87
5.1 Reduced Test Method based on Linear Codes .................. 87
5.2 Simulation Resources .............................................. 96
5.3 Statistical Reduction on the Reduced Test Method ........... 98
5.4 Reduced Test Method based on Rewiring ...................... 104
5.5 Probabilistic Analysis ............................................ 107
5.6 Reliability - An Introductory Analysis .......................... 119
6 Conclusions and Future Work

Bibliography

Appendix A
A1 CONVERT User's Guide
A2 CONVERT Listing
A3 CONVERT Input and Output Files

Appendix B
B1 LAST User's Guide
B2 LAST Listing
B3 LAST Output File

Appendix C

Appendix D
D1 Subroutine to Simulate LFSR (fsimm)
D2 fsimm.ou Input Data
D3 fsimm.ou Output Data

Abstract
List of Figures

Figure 1.1 - General Test Configuration ........................................... 2
Figure 1.2 - General BIST Configuration ........................................... 2
Figure 1.3 - Simple Circuit with stuck-at 1 Fault ................................. 6
Figure 1.4 - Bridge Fault (a) Lines Shorted, (b) TTL Equivalent (Wired AND) 6
Figure 1.5 - Path Sensitization Example Circuit ................................... 9
Figure 1.6 - Scan Design .................................................................... 12
Figure 1.7 - Design with Boundary Scan ............................................. 13
Figure 1.8 - General Form of Polarity Hold SRL .................................... 14
Figure 1.9 - General Form of an LSSD Subsystem with Two System Clocks ... 15
Figure 1.10 - Typical Scan-Path Flip-Flop ........................................... 16
Figure 1.11 - Typical Polarity-Hold Addressable Latch ............................ 17
Figure 1.12 - General Form of the Scan/Set Logic .................................. 18
Figure 1.13 - 4-bit BILBO Structure .................................................. 18
Figure 1.14 - BILBO Configurations. (a) Parallel Register mode (B_1B_2=10), (b) Shift Register mode (B_1B_2=01), (c) LFSR mode (B_1B_2=11) .. 19
Figure 2.1 - Shift Register. (a) Basic Shift Register, (b) Shift Register with Simple Feedback (Cyclic). ......................... 24
Figure 2.2 - Maximum Length 3-Stage Shift Register ................................ 25
Figure 2.3 - An n-Stage LFSR ............................................................... 26
Figure 2.4 - LFSR. (a) Primitive f(x), (b) Non-Primitive f(x) .................... 29
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>An n-Stage Modular LFSR</td>
<td>32</td>
</tr>
<tr>
<td>2.6</td>
<td>A (10,8)LFSR</td>
<td>37</td>
</tr>
<tr>
<td>3.1</td>
<td>General Test Response Analysis</td>
<td>38</td>
</tr>
<tr>
<td>3.2</td>
<td>General Concept of ONEs Count</td>
<td>40</td>
</tr>
<tr>
<td>3.3</td>
<td>Non-Uniform Masking Distribution of ONEs Count</td>
<td>41</td>
</tr>
<tr>
<td>3.4</td>
<td>General Concept of Transition Count Method</td>
<td>43</td>
</tr>
<tr>
<td>3.5</td>
<td>General Concept of Parity Check</td>
<td>44</td>
</tr>
<tr>
<td>3.6</td>
<td>General Concept of Walsh Spectra Testing</td>
<td>46</td>
</tr>
<tr>
<td>3.7</td>
<td>General Concept of Multiple-Parity</td>
<td>47</td>
</tr>
<tr>
<td>3.8</td>
<td>LFSR as Signature Analyzer</td>
<td>48</td>
</tr>
<tr>
<td>3.9</td>
<td>MISR. (a) Standard LFSR, (b) Modular LFSR</td>
<td>50</td>
</tr>
<tr>
<td>3.10</td>
<td>LFSR. (a) Fault-Free, (b) Fault without Aliasing, (c) Fault with Aliasing</td>
<td>51</td>
</tr>
<tr>
<td>3.11</td>
<td>LFSR with Error Added to a Fault Free Input</td>
<td>54</td>
</tr>
<tr>
<td>3.12</td>
<td>LFSR with Error Input</td>
<td>54</td>
</tr>
<tr>
<td>3.13</td>
<td>Markov Process. (a) LFSR with f(x) = x^2 + x + 1, (b) State Transition Diagram, (c) State Probability Diagram</td>
<td>55</td>
</tr>
<tr>
<td>3.14</td>
<td>Aliasing Probability for f(x) = x^3 + x^2 + 1 as Function of Length</td>
<td>61</td>
</tr>
<tr>
<td>3.15</td>
<td>Aliasing Probability for f(x) = x^3 + x + 1 as Function of Length</td>
<td>62</td>
</tr>
<tr>
<td>4.1</td>
<td>Basic Circuit Partitioning. (a) Original, (b) Partitioned</td>
<td>64</td>
</tr>
<tr>
<td>4.2</td>
<td>Partitioning Testing. (a) Additional Inputs, (b) Using Multiplexers</td>
<td>65</td>
</tr>
</tbody>
</table>
Figure 4.3 - Partitioning Based on Multiplexers .......................... 66

Figure 4.4 - Circuit Diagram of TTL SN54/74LS181 .......................... 67

Figure 4.5 - Partitioning of the SN54/74LS181 .......................... 68

Figure 4.6 - Sensitized Testing. (a) L_i Functions, (b) H_i Functions .......... 69

Figure 4.7 - Two-Stage Adder ................................................. 70

Figure 4.8 - Simple Circuit for Verification Testing .......................... 71

Figure 4.9 - Merging Example. (a) Circuit, (b) Non-Adjacency Graph ........ 76

Figure 4.10 - Non-Adjacency Graph. (a) After First Deletion,
(b) After Second Deletion .................................................. 79

Figure 4.11 - Segmentation. (a) Original Circuit Section,
(b) Segmented Circuit Section, (c) Segmented Graph .............. 84

Figure 4.12 - Segmentation. (a) Original Graph, (b) Partitioned ............ 86

Figure 5.1 - Flow Graph to Design an (n,k)LFSR Implementing the
Reduced Test Method Based on Linear Codes .......................... 89

Figure 5.2 - Test Circuit to Illustrate the
Reduced Test Method based on Linear Codes .......................... 91

Figure 5.3 - Number of Test Patterns at Different Fault Coverage (c5315.isc) 102

Figure 5.4 - Number of Test Patterns at Different Fault Coverage (c880.isc) 104

Figure 5.5 - Comparison of Probability of Detection Between Random and
Pseudo-Random Processes .................................................. 111
Figure 5.6 - Comparison of Probability of Detection \( p_d \) for

Different Number of Faults ........................................... 112

Figure 5.7 - Comparison of Probability of Detection \( p_d \) for

Different Detection Probabilities \( p \) .............................. 113

Figure 5.8 - Test Set Representation .................................... 114

Figure 5.9 - Probability of Detection Using Pseudo-Random Model,

where \( p = 5/256 \) ................................................... 115

Figure 5.10 - Speed-up Corresponding to Figure 5.9 .................. 116

Figure 5.11 - Failure Density Function for c7552.isc ................. 120

Figure A1 - c17.isc (ISCAS85) ......................................... 152

Figure A2 - c17.tes (TESS) .............................................. 152
List of Tables

Table 2.1 - Number of Primitive Polynomials of Degree n ...................... 30

Table 4.1 - Dependence Matrix of Circuit of Figure 4.8 ..................... 72
Table 4.2 - Dependence Matrix of Circuit of Figure 4.9.a ..................... 76
Table 4.3 - Initial Candidates ............................................. 78
Table 4.4 - Candidates After First Selection ................................. 79
Table 4.5 - Candidates After Second Selection ................................. 80

Table 5.1 - Dependence Matrix of Circuit of Figure 5.2 ..................... 92
Table 5.2 - Number of Test Patterns for Circuit of Figure 5.2 ............. 93
Table 5.3 - Reduced Test Method Applied to
           TI SN54/74LS630 for \( p(x) = 1 + x^3 + x^{20} \) ....................... 94
Table 5.4 - Reduced Test Method Applied to c880.isc ....................... 94
Table 5.5 - Reduced Test Method Applied to c5315.isc ..................... 95
Table 5.6 - Fault Coverage for the c7552.isc using the Reduced Test Method
           for up to 1,000,000 Test Patterns ................................. 96
Table 5.7 - Fault Coverage of ISCAS85 Benchmark circuits
           for up to 1,000,000 Test Patterns ................................. 98
Table 5.8 - Statistical Reduction Applied to c5315.isc .................... 100
Table 5.9 - Statistical Reduction Applied to c880.isc .................... 103
Table 5.10 - Reduced Test Method Based on Rewiring Applied to c880.isc ... 106
Table 5.11 - Cumulative Fault Probability for c7552.isc .................. 121
List of Abbreviations

ASIC Application-Specific Integrated Circuit
BILBO Built-In Logic Block Observer
BIST Built-In Self Test
BIT Built-In Test
CUT Circuit Under Test
DFT Design for Testability
IC Integrated Circuit
LFSR Linear Feedback Shift Register
LSSD Level-Sensitive Scan Design
MISR Multiple-Input Shift Register (LFSR)
ROM Read Only Memory
SRL Shift-Register Latch
VLSI Very Large Scale Integration
1. INTRODUCTION

The advances in *Very Large Scale Integration (VLSI)* technologies offer *Integrated Circuits (ICs)* with low power consumption, high speed, high reliability and improved performance. These advantages are due to the ability to integrate thousands of devices and connections on a chip. Unless the chips can be tested economically, these advantages may be overcome by the IC final price.

The tests applied to the chip should detect all the possible faults that could prevent the chip from proper operation. In other words, the tests should verify whether the circuit has been manufactured properly and performs the intended functions.

The rapid increase in circuit density and complexity is making testing more difficult and expensive. Testing is becoming a significant portion of the overall cost of *VLSI* circuits, and now accounts for nearly a third of the costs [Levitt 92].

The increase in density and complexity of *VLSI* circuits allows some additional circuitry in the chip to be devoted to testing. This concept is known as *Built-in Test (BIT)* or *Built-in Self Test (BIST)*. *BIST* can perform on-chip test generation and test verification with adequate fault coverage.

The general test configuration for a digital circuit (as chip or board) is presented in Figure 1.1, where the *Circuit Under Test (CUT)* is inside the dotted line.
The general BIST organization is shown in Figure 1.2, where the area inside the dotted line represents the IC borders.
BIST alleviates the problems of delivering test patterns to an unpackaged VLSI chip (or wafer) through wafer probes, which cause impedance discontinuities. For most cases, BIST requires that just power supply and clocking signal be supplied to a VLSI structure with BIST to obtain a pass/fail decision signal. It becomes very important since the test equipment based on probes are very expensive, usually above US$ 1 million [Bardell 87].

Unfortunately there are some penalties for the use of BIST. Extra chip area is the most obvious, which can reach up to 10%, plus performance degradation that occurs when the critical timing path of the original circuit is modified to allow BIST.

Placing the testing circuitry inside the IC is not enough to solve the testing problems. The density of the VLSI circuits and their number of pins causes the exhaustive testing time to be prohibitively long. Instead of using exhaustive testing, pseudo-exhaustive testing methods had been proposed. Unfortunately, pseudo-exhaustive testing time still remains long.

This work explores the existing pseudo-exhaustive testing methods, and presents new approaches and methodologies to improve them. Also, few incomplete portions of the existing methods are completed.

The reduced testing method to be presented is based on pseudo-exhaustive testing. In this proposed method, all advantages of pseudo-exhaustive testing are maintained, while applying only a fraction of all test vectors. Starting with a seed (initial state), for each test pattern generated by LFSR, the CUT is tested (simulated), and the detected faults are marked. This process is maintained until all
faults are detected. Simulation can be carried out for different designs of LFSRs and the best result is selected to be implemented in the final circuit.

The remaining of this chapter is aimed to provide general background of BIST. It also introduces the terminology and testing concepts, and an overview of design for testability (which is used as the base for BIST).

1.1 BACKGROUND

Digital systems testing deals with detection of hardware faults, which are improper states of a system resulting from failure of components, physical interference from environment, operator errors or even incorrect design [Avizienis 75]. An error is the incorrect output signal caused by a fault when the input test patterns are applied. Fault detection involves the application of test patterns that can produce errors caused by some specific set of faults.

Faults can be classified as permanent and transient. Permanent faults represent physical defects that are stable and continuous, and generally due to irreversible physical damages in the system. Transient faults, on the other hand, are present occasionally and are due to unstable hardware or temporary environmental conditions. Unfortunately, there are no reliable means of detecting transient faults, since they may disappear when a test is applied. Those faults will not be considered in this work.
Faults can also be classified as logical or parametric. A logical fault causes the logic function of a circuit element to be changed. A parametric fault can corrupt the magnitude of a circuit parameter, such as speed (delay), current or voltage. Circuit delays can affect the timing operation of the circuit, which may cause hazards or circuit races. Delay faults require testing the circuit "at-speed", in which the clocking of the circuit during test must be the same as in its normal operation.

Just like any other analysis, fault analysis requires modeling. Modeling helps to generate tests and helps to evaluate the test quality (defined in terms of fault coverage). A fault model must be simple to analyze and closely represent the behavior of physical faults in the circuit.

The most common fault model used is the single stuck-at fault, or just the stuck-at fault. Stuck-at fault is the most effective and widespread fault model in use. This model assumes that the fault affects only the gates' interconnections. Each line can be stuck-at 1 (s-a-1) or stuck-at 0 (s-a-0), if the line always assumes logical value 1 or 0, respectively.

Figure 1.3 shows a simple circuit with a stuck-at 1 in the output of the gate G1. The effect of this fault (for this circuit) and a stuck-at 1 at the output of gate G3 is indistinguishable at the output of the circuit. Any test that detects one of them will detect the other. Such faults can be "collapsed", thus reducing the total number of faults to be detected.
Faults in which two lines are shorted together are called bridging faults, and a wired logic occurs. The effect of this fault is dependent on the circuit technology. For instance, Figure 1.4.a shows a bridging fault at the output of gates G1 and G2. If TTL technology had been used, the effect of this fault would be like the wired-AND, illustrated as gate G3 on Figure 1.4.b.

Figure 1.3 - Simple Circuit with Stuck-at 1 Faults.

Figure 1.4 - Bridge fault  (a) Lines Shorted  (b) TTL Equivalent (Wired-AND).
A stuck-open fault in a CMOS logic is any failure that permanently leaves a series connection in a non-conducting state, or a high impedance state. In this case, a logic fault prevents one network from conducting when the other is in a non-conductive state. It has been shown [Reddy 84] that the test set to detect all stuck-at faults together with appropriate initializing inputs can detect also the stuck-open faults.

Stuck-on fault in a CMOS logic is any failure that permanently leaves a transistor (or network) in a conducting state. If one network is permanently on a conducting state and the other goes to a conducting state, a conducting path between $V_{dd}$ and $V_{ss}$ will be created. The logic at the output of the faulty gate would depend on the relative resistance of the conducting networks to the output node. It may be necessary to monitor the steady-state supply current to detect this fault.

Although faults can be modeled as multiple stuck-at faults, unidirectional faults and others, for most practical purposes logical faults can be successfully modeled using stuck-at model. It has been shown that mostly other combinational faults are detected with a high coverage (99%) when test sets based on the single stuck-at model are used [Bardell 87][McCluskey 88]. Also, depending on the circuit, there may be undetectable faults, which can not be detected and are not considered in this work. For convenience, in the remaining of this work, "stuck-at" faults will be used to represent "single detectable stuck-at" faults.
The stuck-at model is specifically suitable for use with algorithmic test
generation, such as D-algorithm [Roth 67], PODEM [Goel 81] and FAN [Fujiwara
83] among others.

Most test generation algorithms work on the principle of path sensitization.
These algorithms try to find an input vector (or sequence of vectors) that will
sensitize a path from the fault location (through the circuit) until the output. The
sensitization is performed by assigning proper values to the gates on the path. The
best known algorithm for test generation is the D-algorithm. This algorithm will
generate a test for any fault, if such test exists.

To illustrate the path sensitization, consider the circuit of Figure 1.5 as an
example. If there is a stuck-at-0 on line $X_3$, the method assigns a 1 for line $X_3$
(different logic values for faulty case). The other input of gate $G_3$ must be 1, so that
the fault can propagate, then $X_2$ is 0. The output of $G_1$ must be 0 to make the fault
visible at the output of $G_4$, and since $X_2$ is 0, $X_1$ can be any logic value (denoted as
d). The input test vector found to detect $X_3$ is d01.
On the D-algorithm, the path sensitization is divided into two phases. In the first phase, a path from the fault location to the output is searched (sensitized path). The second phase searches for values at the input that realize the desired path, or justify it. Also, the D-algorithm tries to sensitize more than one path at the same time, to speed up the process.

To overcome the inefficiency of the D-algorithm for circuits with many exclusive-OR gates, the PODEM algorithm has been proposed, which reduces the backtracking of the D-algorithm. Another algorithm, FAN, which uses more information of the topology of the circuit, reduces further the backtracking and performs faster than the PODEM.

After the test vector is obtained for a given fault, the circuit is fault simulated to verify if additional faults are detected by the given vector. Fault simulation is also
used to verify the quality of a set of tests in terms of fault coverage (faults detected by a test set).

In the fault simulation process, after determining the fault-free signal values of the circuit, the signal values of the faulty circuit are found for one fault at a time (serial fault simulation). It is also possible to simulate a number of faults at the same time (parallel fault simulation method), which depends on the word size used by the host computer. An even larger number of faults can be simulated using the deductive method. In this method a list of sensitized faults is associated with each line of the circuit. Simulation is to deduce or propagate this list of faults along the circuit, accordingly to the signals applied at the input. Unfortunately, it requires large computational memory.

A specific fault in a combinational circuit can be tested with a single vector, but a sequential circuit usually requires usually a sequence of vectors, which makes the test more complex. No known sequential test generation algorithm can provide satisfactory performance [Agrawal 88], although promising algorithms are being developed [Ghosh 92].

Since test generation for sequential circuits is complex and not completely effective, design-for-testability methods (such as scan design) becomes viable. Using these methods, sequential circuits can be tested as combinational circuits [Williams 83].
1.2 DESIGN FOR TESTABILITY

Design for testability (DFT) is a set of design techniques based on the principles of controllability and observability that produce testable designs. These techniques reduces the cost of test generation, and improves the quality of the test.

If the values in all latches can be controlled to any specific value, and if they can be easily observed, then sequential circuits can be tested as combinational circuit. A control signal can switch the elements from their normal mode of operation to a mode that makes them controllable and observable.

The abilities to control and observe any value in the circuit can be implemented using Scan Design. Scan design methods require that memory elements in the circuit are able to be changed between normal and scan mode [Bardell 87].

Figure 1.6 shows how flip-flops are connected in a scan design. The multiplexer (or switch) at the inputs of the flip-flops selects either D, for normal circuit operation, or SCAN-IN, for scan mode. In scan mode, the flip-flops are chained together in the form of a shift register. In this mode, any arbitrary bit sequence can be applied to the SCAN-IN line, thus setting each latch to a desired value. The circuit can then be switched to normal mode and tested with the values on the latches and any pattern on the primary inputs. After switching back to scan mode, the resulting values on the latches can be shifted out and observed at the SCAN-OUT line [Agrawal 88].
One sequence can be scanned in at the same time that another sequence is scanned out. Long chains of flip-flops may be broken into more than one chain, and the chains formed may be scanned concurrently.

Scan design can be extended to boundary scan, which implements scan at board level, as is shown in Figure 1.7. It eases the problems of board testing, specially those using Surface-Mount Technology (SMT), since the values of all IC input and output pins can be controlled and observed. A shift-register latch adjacent to each component pin makes the board behave as the flip-flops in the regular scan design [Williams 83].
There are four basic scan-design structures; Level-Sensitive Scan Design (LSSD), Scan-Path, Scan/Set and Random access.

The most used and best documented structured scan design is the Level-Sensitive Scan Design (LSSD), introduced by [Eichelberger 78] and it is standard at IBM [Williams 83]. Figure 1.8 shows the Shift Register Latch (SRL) used in LSSD as the basic memory element. The SRL consists of two latches L₁ and L₂. When Shift Clock A and Shift Clock B are set to 0, L₁ operates like a polarity-holder latch, with System Data D and System Clock C. When operating as an SRL, the Scan Data I (from the previous stage) is applied into the L₁ by the Shift Clock A. Once stabilized, the Shift Clock A is turned off and the Shift Clock B is turned on to capture the value of L₁ into L₂. Each cycle of A and B moves the data one step into the shift register, and to ensure proper operation of the shift register, A and B must be nonoverlapping.
Figure 1.8 - General Form of a Polarity-Hold SRL.

Figure 1.9 shows a double-latch approach, which is slightly different of the single-latch approach presented in Figure 1.8. In the double-latch, L₁ and L₂ are used as system latches, and the system output is taken from L₂. This design technique requires that the Shift B Clock be also a system clock (or C₂ Clock). This design also requires nonoverlapping shifts (or clocks).
The Scan-Path structure (very similar to LSSD) also implements storage elements as stages of a shift register for scanning test data in, and test response out. It was introduced in [Funatsu 75] and it is the methodology used by Nippon Electric Co. (in Japan) [Williams 83]. The stages are built with raceless D-type flip-flops. Figure 1.10 shows a typical scan-path flip-flop, where normal operation uses \textit{Clock 1} and scan mode uses \textit{Clock 2}. 
During system operation, \textit{Clock 2} is held at 1. With \textit{Clock 1} at logical 0, the value at \textit{System Data} is captured by FL\textsubscript{1}, and with \textit{Clock 1} at logical 1, the value held in FL\textsubscript{1} is captured by FL\textsubscript{2}. In a similar fashion, scan operation uses \textit{Clock 2}, while \textit{Clock 1} is always at 1.

Another structured scan design is the Random-Access scan, used at Fujitsu [Ando 80]. Each element is addressed individually, in a fashion similar to random-access memories. Figure 1.11 shows the addressable polarity-hold latch used in random-access scan. During normal operation \textit{Scan Clock} is inactive and the value of \textit{Data} is latched upon the action of \textit{Clock}. During test operation, the value on the \textit{Scan Data In} is gated onto the latch by the \textit{Scan Clock} if \textit{X-address} and \textit{Y-address} lines are selected simultaneously. The output of the latches, during test operation, can have a common circuit output, since only one latch is active at a time.
The last structured scan design is the Scan/Set logic, which was introduced by Sperry-Univac [Stewart 77], is presented in Figure 1.12. An auxiliary Scan/Set register is appended to the original circuit for either scanning out logic values or setting values into the circuit storage or both. Since the original circuit is unchanged (except for the added control and observation points), test for the original circuit is more complex than using other techniques. However, the main circuit elements are not restricted to any type of flip-flop.

The added register can capture data in parallel from the main circuit for serial unload on its scan-out port. It can also be serially loaded into its scan-in port for parallel application to test points in the main circuit.
An interesting design technique that integrates scan design with signature analysis (to be presented in chapter two) is the *Built-in Logic Block Observer* (BILBO). The BILBO facilitates BIST and external testing, while minimizing overhead due to flip-flops. Figure 1.13 gives the form of a 4-bit BILBO.
A BILBO has the control lines $B_1$ and $B_2$ that determines its mode of operation. Figure 1.14.a shows the BILBO set to parallel-in parallel-out register ($B_1B_2 = 10$), which is the configuration for normal circuit operation. When $B_1B_2 = 01$, as in Figure 1.14.b, the BILBO takes the form of a shift register.

In the third mode ($B_1B_2 = 11$), the BILBO acts as a *Linear Feedback Shift Register* (LFSR) with multiple inputs, as in Figure 1.14.c. If the inputs are set to logical 0, the BILBO works as a pattern generator. When $B_1B_2 = 00$, the BILBO is reset.

![Diagram](image)

**Figure 1.14 - BILBO Configurations** (a) Parallel Register mode ($B_1B_2 = 10$), (b) Shift Register mode ($B_1B_2 = 01$), (c) LFSR mode ($B_1B_2 = 11$).
1.3 TEST GENERATION METHODOLOGIES

Testing of a circuit can be performed off-line or on-line. Off-line (or non-concurrent) testing is performed to ascertain the proper structure or function of the circuit, usually performed by the manufacturer. If a procedure to test the system is performed as a task in the job stream of the system while other tasks are ongoing, the test is said to be on-line (or concurrent). Most work on testing of digital circuits is focused on off-line testing.

The best known categorizations of off-line testing are functional testing, prestored testing, exhaustive testing and random testing.

Functional testing is a verification of the intended function of the circuit. But to ensure that the circuit is fault-free, it is not enough to show that the intended function has been performed correctly. It is also necessary to verify that no unintended function is additionally performed.

In the Prestored testing a limited set of test patterns obtained from test pattern generation can be stored in a Read-Only Memory (ROM). Unfortunately the ROM becomes rather large for complex circuits, where its size is proportional to the number of test patterns.

Exhaustive testing implies the application of all $2^n$ tests, where $n$ is the number of circuits inputs of the Circuit Under Test (CUT). Applying all possible test vectors to the CUT guarantee that all stuck-at faults will be detected (except the redundant faults). The disadvantage of this method is the time it takes to test large circuits. To
illustrate, consider a circuit with 64 inputs, with tests applied at the rate of 1 test per nanosecond would require about 585 years to exhaustively test the circuit. Under the same considerations, a circuit with 32 inputs would require just 9 seconds.

Random testing applies only a (random) subset of the exhaustive testing. Actually the process should be called pseudo-random because the test sequence is predictable and repeatable. The number of tests in this subset can be obtained by probabilistic measures of the circuit (based on analytical methods), which becomes another problem itself. The size of the subset is also restrained by the economically allowable test time.

Chapter Two introduces the LFSR, its modeling and use as a test pattern generator.

Chapter Three presents the common used methods of test response verification and their respective aliasing probabilities. It also shows the use of LFSR as signature analyzer with a deeper insight in its aliasing.

On Chapter Four the pseudo-exhaustive methods are presented, with their advantages and limitations. An algorithm for minimum covering for verification testing is presented. This chapter also presents programs developed to obtain circuit dependence and minimum covering for verification testing.

The reduced test methods developed in this research are presented in Chapter Five, including their analysis of aliasing, fault detection probability and respective extensive simulations. It is also included in this chapter an introduction on reliability issue.
In the reduced test method, instead of generating all the patterns for each cone of logic of the CUT, the patterns are sequentially generated only until all faults are detected. A group of possible LFSR configurations is simulated along with the CUT, and the configuration resulting in the shortest testing time is selected for actual circuit implementation. For each test pattern generated by the LSFR, the CUT is fault simulated, and the detected faults are marked. This is repeated until all faults have been detected.

Programs to fault simulate the CUT have been modified to permit the use of LFSR as test pattern generator. Other programs have been developed to automate pseudo-exhaustive testing.

This work is concluded in Chapter Six which analyzes the results and suggests future work.
2. LINEAR FEEDBACK SHIFT REGISTERS

Linear Feedback Shift Registers can be used as test pattern generators (Stimulus Generators) and can perform response analysis. Among the test pattern generators and response analysis, LFSR is the most widely used.

Some LFSRs have their characteristic polynomial primitive, and can generate exhaustive test patterns. Some others may have their characteristic polynomial based on Linear Codes, and those are suitable for verification testing, which is a pseudo-exhaustive technique.

2.1 SHIFTER REGISTERS

A shift register is a set of storage elements connected so that the state of each element is shifted to the next element (in a certain direction) in response to a shifting signal. Figure 2.1.a shows a three-stage shift register, where each stage is a storage element (such as flip-flop D).

Figure 2.1.b shows a simple feedback shift register, where the output of the last stage is fed into the input of the first stage. In this figure and in the remaining of this work, for simplicity, the flip-flops will be represented by a single box, without clock lines.
Taking [001] as the initial state in the shift register of Figure 2.1.b, the shift register will go to state [100] after a clock cycle, then [010] and will come back to initial state [001], or

\[
S_0 = [001] \\
S_1 = [100] \\
S_2 = [010] \\
S_0 = [001] \tag{2.1}
\]

If, instead, the connections of the shift register are chosen according to Figure 2.2, where the symbol + (or \(\oplus\)) denotes Exclusive-Or (modulo \(2^1\) addition), seven different states would have been obtained, accordingly,
$S_0 \quad [001]$
$S_1 \quad [100]$
$S_2 \quad [010]$
$S_3 \quad [101]$
$S_4 \quad [110]$
$S_5 \quad [111]$
$S_6 \quad [011]$
$S_0 \quad [001]$

(2.2)

Figure 2.2 - Maximum Length 3-stage Shift Register.

It can be seen that the next state of stage 0 is the present state of stage 1, and the next state of stage 1 is the present state of stage 2. In this shift register, the next state of stage 2 is \textit{state (stage 1)} \oplus \textit{state (stage 0)}, which can be verified by the state sequence given by (2.2). This case corresponds to a maximum length cycle that can be obtained with a three-stage feedback shift register.
2.2 SHIFT REGISTER WITH LINEAR FEEDBACK

A digital circuit is linear if it is made of unit delays (latches), modulo-2 adders (Ex-OR's) and modulo-2 multipliers (shift operation). A linear digital circuit has the property that its response to a linear combination of the inputs preserves the principle of superposition.

The principle of superposition states that the response of a linear network to a linear combination of stimuli is the linear combination of the response of the network to the individual stimuli. The digital circuits require that the initial state of the storage elements should be zero, to preserve the principle of superposition. This initial condition is equivalent to the DC-offset on a analog circuit.

A shift register with linear feedback is a linear digital circuit and is called Linear Feedback Shift Register (LFSR). Figure 2.3 illustrates a standard configuration of a LFSR. The present state of the n-stage LFSR (at time \( t \)) is \([x_0(t), x_1(t),...,x_{n-1}(t)]\).

The next state of such LFSR (or time \( t+1 \)) is \([x_0(t+1), x_1(t+1),...,x_{n-1}(t+1)]\).

Figure 2.3 - An n-stage LFSR
The relation between the present and the next state of the LFSR is represented by

\[
\begin{bmatrix}
X_0(t+1) \\
X_1(t+1) \\
X_2(t+1) \\
\vdots \\
X_{n-2}(t+1) \\
X_{n-1}(t+1)
\end{bmatrix}
= \begin{bmatrix}
0 & 1 & 0 & \ldots & 0 & 0 \\
0 & 0 & 1 & \ldots & 0 & 0 \\
0 & 0 & 0 & \ldots & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \ldots & 0 & 1 \\
1 & h_1 & h_2 & \ldots & h_{n-2} & h_{n-1}
\end{bmatrix}
* \begin{bmatrix}
X_0(t) \\
X_1(t) \\
X_2(t) \\
\vdots \\
X_{n-2}(t) \\
X_{n-1}(t)
\end{bmatrix}
\]

(2.3)

where \( h_i \) (\( 0 \leq i \leq n-1 \)) is either 1 or 0, depending on the existence or absence of the connection \( h_i \).

Equation (2.1) can also be represented in vector format as

\[
X_{(t+1)} = C * X_{(t)}
\]

(2.4)

where \( C \), the \((n-by-n)\) companion matrix [Golomb 82], relates the present \((x_0)\) and the next state \((x_{0+1})\) of the LFSR.

If \( x_0 \) is the initial state of the LFSR, then the sequence of states, during successive clock cycles would be

\[
X_0, \ C*X_0, \ C^2*X_0, \ C^3*X_0, \ \ldots
\]

(2.5)

and the \( i \)th state value of the LFSR would be \( C^i x_0 \).
The characteristic polynomial $f(x)$ of the LFSR is defined as the determinant of $C - Ix$, or

$$f(x) = |C - Ix| \quad (2.6)$$

where $I$ is the $n$-by-$n$ identity matrix

This can be rewritten as

$$f(x) = 1 + h_1x + h_2x^2 + \ldots + h_{n-1}x^{n-1} + h_n \quad (2.7)$$

Given a LFSR, its characteristic polynomial can be obtained by inspection. Also, from the characteristic polynomial, the corresponding LFSR can be easily obtained, given the simplicity of such equation.

The smallest $T$ such that $C^T = I$ is the period (or maximum cycle length) of the LFSR, where in this case $C^Tx_0 = x_0$. It has been proven [Bardell87] that the period $T$ is the smallest integer for which $f(x)$ divides the polynomial $1 + x^T$.

### 2.2.1 PRIMITIVE POLYNOMIALS

If $T = 2^n - 1$, then the sequence generated by the LFSR is called maximum length sequence, or m-sequence. The characteristic polynomial of the maximum length sequence is known as primitive polynomial. Only primitive polynomials can generate the m-sequence, or maximum length sequence, which corresponds to all $2^n$
test vectors, except the all-zero vector. For primitive polynomials, the smallest integer for which f(x) divides $1+x^{T}$ is $T=2^{n}-1$. In other words, a polynomial f(x) of degree $n$ is primitive if it divides $1+x^{2^{n}-1}$.

To clarify the differences between primitive and non-primitive polynomials, consider Figure 2.4.a which shows a LFSR implementing a primitive polynomial of degree three. It can be seen that, except for the all-0 state, the LFSR cycles through all other states, and the period is $2^{3}-1$, or 7.

---

Figure 2.4 - LFSR  (a) Primitive f(x), (b) Non-Primitive f(x).

The characteristic polynomial of the LFSR of Figure 2.4.b is not primitive polynomial. It can be seen that depending on the initial condition (called seed of the LFSR), the LFSR will have different sequences (four in this case), each having a different period.
The number of primitive polynomials of degree \( n \) is given by [Golomb82]

\[
\lambda_2(n) = \frac{\phi(2^n - 1)}{n}
\]  (2.8)

in which the Euler \( \phi \)-function is defined as

\[
\phi(i) = i \prod_{p | i} \left( 1 - \frac{1}{p} \right)
\]  (2.9)

where \( p \) corresponds to the relative primes of \( i \) (prime numbers which divide \( i \)). Table 2.1 tabulates \( \lambda_2(n) \) for values of \( n \) up to 16.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( 2^n )</th>
<th>( \lambda_2(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
<td>48</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
<td>60</td>
</tr>
<tr>
<td>11</td>
<td>2048</td>
<td>176</td>
</tr>
<tr>
<td>12</td>
<td>4086</td>
<td>144</td>
</tr>
<tr>
<td>13</td>
<td>8192</td>
<td>630</td>
</tr>
<tr>
<td>14</td>
<td>16384</td>
<td>756</td>
</tr>
<tr>
<td>15</td>
<td>32768</td>
<td>1800</td>
</tr>
<tr>
<td>16</td>
<td>65536</td>
<td>2048</td>
</tr>
</tbody>
</table>
As an example, consider \( n \) equals to 6. Then

\[
\phi(2^6-1) = \phi(63) = 63(1-1/3)(1-1/7) = 36
\]

(since the relative primes of 63 are 3 and 7),

\[
\lambda_2(6) = 36/6 = 6.
\]

Given a polynomial, tests must be performed to determine if the polynomial is primitive or not. The first test rejects all trial polynomials with all even exponents, because such polynomials are square and can be reduced.

A second test checks if the trial polynomial has an odd number of terms. One of which is constant. If one of terms is not constant, the polynomial can be simplified (\( x \) is a factor). If the number of terms is even, \( f(x) \) can be divided by \( x+1 \). The last test requires that the characteristic polynomial of the trial polynomial divides \( x^k + 1 \), where the smallest \( k \) is \( 2^n-1 \). This test becomes difficult for large values of \( n \), for instance, for \( n=20, T=1,048,576 \). Then \( x^{1048576} + 1 \) would have to divide \( f(x) \).

### 2.2.2 MODULAR LFSR

Besides the standard LFSR presented in Figure 2.3, LFSRs can also be arranged as in the Figure 2.5, which is known as modular LFSR. In the standard LFSR, the Ex-OR elements are placed outside the chain of flip-flops, and could be implemented by multiple input Ex-OR gates. The modular LFSR has the Ex-OR elements placed between the flip-flops in the chain. The modular LFSR is more often used as a signature analyzer.
Figure 2.5 - An $n$-stage Modular LFSR.

The companion matrix $D$ of the modular LFSR, which relates the present and the next state of the LFSR, is

$$D = \begin{bmatrix}
0 & 0 & 0 & \ldots & 0 & 1 \\
1 & 0 & 0 & \ldots & 0 & h_1 \\
0 & 1 & 0 & \ldots & 0 & h_2 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \ldots & 0 & h_{n-2} \\
0 & 0 & 0 & \ldots & 1 & h_{n-1}
\end{bmatrix} \tag{2.10}$$

and

$$X_{(t+1)} = D \ast X_{(t)} \tag{2.11}$$

It can be seen that the companion matrix $D$ is the transpose of the companion matrix $C$ of the standard LFSR. The characteristic equation of the modular LFSR
should be the same as the standard LFSR. Given a characteristic equation, the respective LFSR can be implemented as modular or as a standard LFSR.

Define the reciprocal polynomial \( f'(x) \) of \( f(x) \) (of degree \( n \)) as

\[
f'^*(x) = x^n f(1/x)
\]  

If \( f(x) \) is a primitive polynomial, \( f'(x) \) also would be. The sequence generated by the LFSR implementing \( f(x) \) is the reverse sequence generated by \( f'(x) \).

From the definition, an LFSR implementing a primitive polynomial is able to generate all \( 2^n - 1 \) different binary states (except the all-zero state). If the all-zero state must be generated, after generating all the other states, the LFSR could be reset, thus producing such a state.

Another approach is the use of an LFSR having an extra stage (also implementing a primitive polynomial), and taking only \( n \) bits of this LFSR. The \( n \)-bit all-zero state is present in the \( 2^{n+1} - 1 \) patterns generated by the new \( n+1 \) stages LFSR.

2.3 LINEAR CODES

The LFSR with characteristic polynomial based on Linear Codes is used as test pattern generator in the verification testing, which is one of the pseudo-exhaustive testing techniques. It is also used in the reduced test method to be presented in Chapter Five.
The theory of Linear Codes applied to LFSR is based on the work of [Wang85b] and [Wang86a].

Equation (2.5), which gives the characteristic polynomial of a LFSR, takes into account only the coefficients of \( x \), not the actual values. By the same manner, the states of the LFSR can be represented by a polynomial \( S(x) \) of degree \( n-1 \), where \( n \) is the number of stages. For instance, if the initial state of a 4-stage standard LFSR is 1011, then \( S(x) = x^3 + x + 1 \).

The arithmetic used to manipulate those polynomials is the arithmetic of the coefficients, a linear algebra over the field of two elements (0 and 1). The laws of the ordinary algebra are valid, except that operated over modulo-2 (such as 0+0=0, 0+1=1, 1+0=1 and 1+1=0), so that \( (x^2 + 1) + (x^2 + x) \) would be \( (x + 1) \), not \( (2x^2 + x + 1) \). Also \( (x + x) \) is the same as \( (x - x) \).

It has been proven [Hsiao77] that if the initial state of the LFSR is \( S_0(x) \), then the state \( S_i(x) \) (after \( i \) shifts) is given by

\[
S_i(x) = x^i S_0(x) \mod f(x). \tag{2.13}
\]

where the notation
\[
a = b \mod c
\]
means that the residue of \( a/c \) is the same as the residue of \( b/c \).

Since after \( T \) shifts (where \( T \) is the period of the LFSR) the LFSR returns to its initial state, the period of a LFSR can be seen also as
Define an \((n,k)\)LFSR as a group of non-primitive LFSR in which the period is \(2^k\) (where \(k < n\)). Let \(f(x) = g(x) \cdot p(x)\), where \(p(x)\) is any primitive polynomial of degree \(k\) and \(g(x)\), called the generator polynomial, is any polynomial of degree \(n-k\). Theorems 2.1 and 2.2 have been formulated based on the work of [Wang 85b].

\[ S_i(x) = x^r S_i(x) \mod f(x). \quad (2.14) \]

Theorem 2.1:

For a given LFSR with \(f(x) = g(x) \cdot p(x)\), if the initial state \(S_0(x)\) is divisible by \(g(x)\), then the polynomial representing the \(i\)th shift is still divisible by \(g(x)\).

Proof:

Let \(S_0(x)\) be a non-zero polynomial of degree \(n-1\) (or smaller). If the \(i\)th shift is given according to equation (2.13), then there exist a polynomial \(q(x)\) such that

\[ S_i(x) = x^r S_0(x) - q(x) f(x). \quad (2.15) \]

since both \(S_0(x)\) and \(f(x)\) are divisible by \(g(x)\), thus \(S_i(x)\) is also divisible by \(g(x)\).

Q.E.D.

Theorem 2.2:

Any consecutive \(2^k-1\) states of the LFSR represented by \(f(x) = g(x) \cdot p(x)\) are distinct if \(S_0(x)\) is divisible by \(g(x)\).
Proof:

If the states $S_i(x)$ and $S_j(x)$ represent respectively the $i_{th}$ and the $j_{th}$ shifts of the LFSR with initial state $S_0(x)$ resulted in the same value, then

$$x^iS_0(x) = x^jS_0(x) \mod f(x) \quad (2.16)$$

or

$$x^iS_0(x) = x^jS_0(x) \mod g(x)p(x) \quad (2.17)$$

and since $S_0(x)$ is divisible by $g(x)$

$$(x^i-x^j)S_0(x) = 0 \mod g(x)p(x). \quad (2.18)$$

Let $S_0(x) = g(x) \ast a(x)$, where $a(x)$ is any polynomial of degree $k-1$, then

$$(x^i-x^j)a(x) = 0 \mod p(x) \quad (2.19)$$

which can be written as

$$x^i(1+x^{3-i})a(x) = 0 \mod p(x) \quad (2.20)$$
since $a(x)x^i$ and $p(x)$ have no common factor, it implies that $p(x)$ divides $1+x^{i^j}$. Given that $p(x)$ is a primitive polynomial, $T$ is the smallest integer such that $1+x^T$ divides $p(x)$. Then it follows that

\[ i-j = 0 \mod T \quad (2.21) \]

or

\[ i = j \mod T \quad (2.22) \]

Q.E.D.

Consider, as an example the $(10,8)$LFSR of Figure 2.6. For this case $g(x) = 1 + x + x^2$, $p(x) = 1 + x^2 + x^3 + x^4 + x^8$, and $f(x) = g(x)*p(x) = 1 + x + x^4 + x^6 + x^8 + x^9 + x^{10}$. The period is $2^{10}-1 = 255$.

![Figure 2.6 - A (10,8)LFSR](image)

The $(n,k)$LFSR is used as test pattern generator for one of the pseudo-exhaustive testing methods, and is also used as test pattern generator in the reduced test method, in Chapter Five. For a given value of $w$ ($w<k$), there is a value of $k$ such that, any $w$ outputs of the $(n,k)$LFSR can exhaustively generate all $2^w$ test patterns.
3. RESPONSE ANALYSIS

Ideally, the test response of a circuit under test should be compared bit-by-bit to the expected response. Given the large reference storage capacity required, it is usual to capture some statistics, called signatures, of the circuit output, instead of the individual bits themselves. This signature is then compared to the fault-free circuit signature. Figure 3.1 shows the concept of a general test response analysis.

![Diagram](image)

Figure 3.1 - General Test Response Analysis.

Unfortunately, any data compression method loses some information. This loss is usually called aliasing or masking, which is measured by the probability that a faulty circuit will produce the same signature as the fault-free circuit.

Any signature method must be simple, easy to implemented and fast (compared to the speed of the CUT). The method should also have some form of logarithmic compression to minimize the signature storage area. Ideally, the method should also have the lowest possible aliasing.
An important measure of a compression method is how it minimizes aliasing. There are basically two ways to measure the aliasing characteristics of a compression method. The first is to simulate simultaneously the CUT and the proposed compression method, apply the actual test sequence, and determine those faults that cause masking. Although the simulation method is precise, it is time consuming and it is dependent on the circuit under test and the test sequence applied.

Another approach is to compute the proportion of all error sequences that can mask over the fault-free sequence. This method is not so precise, but it is circuit independent, and it is widely accepted to compare different compression methods.

3.1 COMPRESSION METHODS

The basic compression methods are presented in this section with their respective aliasing probabilities. To compare the methods the assumption is made that all possible CUT output responses are equally likely to occur. Even though this is not likely to occur, it provides means to compare the different compression methods.

The basic compaction methods are ONE’s counting, transition counting, parity and the use of LFSR. There are also few modifications on the basic methods, to be presented.
3.1.1 ONE'S COUNT

Figure 3.2 shows the ONE's count framework for a CUT with single output. The signature is the number of ONE's on the test response output of the CUT. The ONE's count requires a counter with at most \( \lceil \log_2(n+1) \rceil \) stages, where \( \lceil x \rceil \) is the smallest integer greater than or equal to \( x \), and \( n \) is the number of bits of the test response (from the CUT).

![Diagram of ONE's Count](image)

Figure 3.2 - General Concept of ONE's Count.

Masking will occur if a faulty circuit produces the same number of ONE's as in the original sequence. Consider that \( r \) ONE's are expected (\( 0 \leq r \leq n \)), then there are \( \binom{n}{r} \) different sequences that can produce \( r \) ONE's, and one of those sequences is the correct sequence. The ratio of incorrect sequences with \( r \) ONE's to the total number of incorrect sequences \( (2^n-1) \) is the masking probability for a n-bit sequence with the ONE's count of \( r \), or

\[
P(M\mid r) = \frac{\binom{n}{r}-1}{2^n-1}
\]  
(3.1)
As can be observed in Figure 3.3, the probability of masking is low when the signature is close to the extremes of its range, but increases rapidly towards the center. When the expected count is either ZERO or \( n \), aliasing is not possible, since there is only one sequence of ZERO's and only one sequence of ONE's.

![Diagram](image)

**Figure 3.3 - Non-Uniform Masking Distribution of ONE's Count.**

To minimize masking, the CUT can be modified such that the ONE's count is close either to its minimum (0) or its maximum (n).

The probability that a given n-bit random sequence will have \( r \) ONE's is

\[
P(r) = \frac{\binom{n}{r}}{2^n}
\]  

(3.2)
The product $P(r)P(M|r)$ can be summed over all values of $r$ to obtain the masking probability $P(M)$, or

$$P(M) = \sum_{r=0}^{n} P(M|r) P(r)$$ \hspace{1cm} (3.3)

which results (from 3.1 and 3.2)

$$P(M) = \sum_{r=0}^{n} \frac{\binom{n}{r} - 1}{2^{n-1}} \cdot \frac{n}{2^n}$$ \hspace{1cm} (3.4)

Using Stirling's formula for factorials (for large $n$),

$$P(M) = (\pi n)^{-\frac{1}{2}}$$ \hspace{1cm} (3.5)

$P(M)$ is the probability that a random sequence will be masked, based on the ONE's count method.

The ONE's count method guarantees detection of single-bit errors, because this error causes the ONE's count to be changed, but masking can occur for other sets of errors.

For CUTs with multiple outputs, one ONE's counter could be placed in each output, which greatly increases the area overhead (increased number of counters). Another approach is the use of a parallel-to-serial conversion, so that the outputs of the CUT are fed into the parallel-to-serial converter, whose output is applied to a
ONE's counter. This idea, unfortunately, requires the additional time to serially apply the signal to the counter, plus the extra size of the counter and the parallel-to-serial converter itself.

3.1.2 TRANSITION COUNT

In the transition count method, the signature is the number of logical transitions (1→0 and 1→0) in the CUT output response. Figure 3.4 presents the transition count framework.

\[
\sum_{i=1}^{n} (r_i \oplus r_{i-1})
\]

(3.6)

where \( r_i \) (1≤i≤n) is the \( i \)th bit of the sequence.
Masking will occur if the faulty circuit produces an output with the same number of transitions as in the original circuit. Following an analysis similar to the ONE's count, it has been shown [Bardell 87] that the aliasing probability for the transition count is the same as ONE's count, or

$$P(M) = (\pi n)^{-\frac{1}{2}}$$  \hspace{1cm} (3.7)

3.1.3 PARITY CHECK

In this method only one bit is obtained as signature of the test response. Figure 3.5 shows the parity check circuit configuration.

![Figure 3.5 - General Concept of Parity Check.](image)

This method, as with the ONE's count, can detect all single-bit errors, and all signatures with an odd number of errors. Half of all n-bit sequences have odd parity and the other half have even parity. There are then \((2^n/2)-1\) sequences with the same
parity as the correct signature, which divided by the total number of wrong signatures results in the probability of masking,

\[ P(M) = \frac{2^n - 1}{2^{n-1}} \approx \frac{1}{2} \] (3.8)

For multiple output circuits, the considerations are similar to the ONE's count method.

### 3.1.4 VARIATIONS OF THE BASIC METHODS

On the presented methods (ONE's counting, transition counting and parity), there are few variations to be considered. One of the variations of the transition count method is the edge counting method. In this method only one of the transitions (either 1→0 or 1→0) is counted.

A variation of the ONE's method is the Walsh spectral coefficients. The general concept of this method is shown in Figure 3.6. The output of the CUT is EX-ORED with the test sequence and applied to a counter. No production system has been reported using the Walsh testing [Bardell 88].
A possible modification of the parity method is the multiple parity. In this method, besides the regular parity checking there are also some additional parity checking. Each additional parity checking may be obtained by ANDing the regular CUT output and one of the CUT inputs. This method allows up to \((n+1)\) parity checkers, where \(n\) is the number of inputs of the CUT. The implementation of this method can be seen in Figure 3.7, where \(x_i (0 \leq i \leq n)\) is a primary input.

The addition of checkers may reduce the aliasing, but it should be noticed that this method corresponds to an overhead to the original \(ONE's\) count.
3.2 SIGNATURE ANALYSIS USING LFSR

The use of LFSR as a signature analyzer has dominated the BIST design because of its cost and simplicity. This method offers some advantages over the previous methods; it is sensitive to the number of ONE's, their position and has a uniform aliasing probability distribution (opposite of the ONE's count or transition count).

The general form of LFSR being used as a signature analyzer is presented in Figure 3.8. The difference between the LFSR as a pattern generator (Figure 2.3) and as a signature analyzer is the addition of an Ex-OR gate at the input of the first stage.
of the LFSR. This additional gate receives the feedback of the LFSR and receives the output of the CUT.

![Diagram](image)

Figure 3.8 - LFSR as Signature Analyzer.

Although Figure 3.8 represents a standard LFSR, similar analysis could be obtained for the modular LFSR. For the LFSR of Figure 3.8, consider

\[
D_t = \begin{bmatrix}
0 \\
0 \\
. \\
. \\
. \\
0 \\
d_t
\end{bmatrix}
\]

(3.9)

as an n-row vector which represents the incoming bits in the LFSR, where \( t \) represents the \( t \)th test vector applied to the CUT. \( d_t \) appears on the last position of \( D_t \) because this position represents the stage \( n-1 \) of the LFSR.
Using the same notation as in Chapter 2,

\[ x_1 = Cx_0 + D_0 \]

\[ x_2 = Cx_1 + D_1 = C^2x_0 + CD_0 + D_1 \]

and in general

\[ X_i = C^iX_0 + \sum_{j=1}^{i} C^{i-j}D_{j-1} \]  \hspace{1cm} (3.10)

where \( x_i \) represents the \( i \)th state of the LFSR, \( C \) is the companion matrix and \( D \) is the output sequence of the CUT.

The analysis performed for a single input LFSR can be extended to a Multiple-Input Signature Register (MISR). The MISR can be made using modular LFSR (easiest approach) or using a standard LFSR, which requires extra Ex-ORs. Figure 3.9.a shows the standard LFSR as MISR, and Figure 3.9.b shows the modular LFSR as MISR.

The \( n \)-row vector representing the incoming bits in the MISR is

\[
D_t = \begin{bmatrix}
d_{t,0} \\
d_{t,1} \\
d_{t,2} \\
\vdots \\
d_{t,n-1}
\end{bmatrix}
\]  \hspace{1cm} (3.11)

where \( d_{t,j} \) is the \( t \)th bit arriving at the \( j \)th stage of the MISR.
3.3 ALIASING IN LFSR

The incoming bits to an LFSR can be represented by a polynomial $P(x)$. In this representation, $P(x)$ enters the LFSR register high-order first. For instance, the input sequence $P = 10001010$ can be represented by $P(x) = x^7 + x^3 + x$, and the bit represented by $x^7$ (first bit of $P$) enters the LFSR first.

The LFSR of Figure 3.10 has the characteristic polynomial $f(x) = x^3 + x^2 + 1$. As in Figure 3.10.a, the output of a fault-free circuit ($P(x) = x^3$), is applied to the LFSR and the signature (final state of the LFSR) is 101, or $x^2 + 1$. On figure 3.10.b
the output of a faulty circuit $P_1(x) = x^4 + x^3$ is applied to the same LFSR and the respective signature is, 010 or $x$, and the error can be detected. Yet another faulty output test sequence, as presented in Figure 3.10.c ($P_2(x) = x^4 + x^3 + x^2 + x + 1$), will produce the same signature as the fault-free circuit, $x^2 + 1$. The last signature (faulty circuit) is exactly the same as the fault-free of Figure 3.10.a, resulting in aliasing.

Figure 3.10 - LFSR (a) Fault-Free, (b) Fault without Aliasing, (c) Fault with Aliasing.
From Figure (3.10)

\[ P(x) = x^3, \]

\[ P_2(x) = x^4 + x^3 + x^2 + x + 1 \]

and \[ E(x) = P(x) + P_2(x) = x^4 + x^2 + x + 1. \]

\( E(x) \) is the error polynomial, which represents an error in the corresponding bit position of the output test response. \( P(x) \) is the error free CUT output and \( P_2 \) is the erroneous CUT response.

It can be verified that the signature can be represented as the remainder of \( P(x)/f(x) \), i.e.

\[ R(x) = P(x) \mod f(x) \] \hspace{1cm} (3.12)

For Figure 3.10.a, equation (3.16) is

\[ R(x) = x^2 + 1 = x^3 \mod (x^3 + x^2 + 1), \]

which is the same as Figure 3.10.c, or

\[ R(x) = x^2 + 1 = (x^4 + x^3 + x^2 + x + 1) \mod (x^3 + x^2 + 1). \]

An undetected fault can occur if

\[ R(x) = P(x) \mod f(x) = P_2(x) \mod f(x) \] \hspace{1cm} (3.13)

which means they have the same signature.
Any undetected error polynomial $E(x)$ must be evenly divided by $f(x)$. In other words, an undetectable error is represented by an error polynomial that is a non-zero multiple of $f(x)$. For Figure 3.10.c

\[
\frac{E(x)}{f(x)} = \frac{1+X+X^2+X^4}{1+X^2+X^3} = X+1
\]

In an output test sequence of $n$ bits, there are $2^n-1$ error polynomials (one of the $2^n$ sequences is the correct sequence). Since all undetectable error polynomials are non-zero multiples of $f(x)$ (of degree $r$) there are $2^n-r-1$ error polynomials (one of them is the correct one). The probability of masking is the number of error polynomials divided by the total number of sequences, or

\[
P(M) = \frac{2^n-r-1}{2^n-1} \quad (3.14)
\]

for large $n$,

\[
P(M) \approx \frac{1}{2^r} \quad (3.15)
\]

Note that based on this equation, adding one stage to a LFSR, halves the aliasing probability.
Up to this point it is considered that all CUT output sequences are equally likely to occur. The aliasing probability is now reevaluated with the assumption that the faults are independent, each with a given fault detection probability.

As presented, \( E(x) = P(x) + P_2(x) \), or \( P_2(x) = P(x) + E(x) \), then a LFSR having an input signal with error can be represented as in Figure 3.11. The error \( E(x) \) is added to a fault-free signal \( P(x) \) to obtain \( P_2(x) \).

![Figure 3.11 - LFSR with Error Added to a Fault-Free Input.](image)

The LFSR with error added to the fault free input can be analyzed by the configuration presented in Figure 3.12. The LFSR can be analyzed with just the error being applied. A \textit{ZERO} input means no error present for that bit, and a \textit{ONE} represents an error.
Based on that, a Markov process can be described using states and transitions. Figure 3.13.b shows the transition diagram for the LFSR of Figure 3.13.a, where the arcs are labeled by the input symbol that results in the associated transition.
By substituting each input by its probability, the probability transition diagram is obtained, as in Figure 3.13.c. The condition \( p \) (probability of occurrence of a fault or fault detection probability) replaces the arriving of \( \text{ONE} \) (an error condition), and \( \text{ZERO} \) is replaced by \( 1 - p \).

The fault detection probability \( p \) is defined as [Saxena 90]

\[
 p = \frac{d_f}{u} \tag{3.16}
\]

where the detectability \( d_f \) of a fault is the number of applied input test patterns from the set \( u \) that detect the fault.

Let \( \pi[t] = (\pi_1[t], \pi_2[t], \ldots, \pi_{2^r}[t]) \) be the state probability vector, where a component \( \pi_i[t] \) corresponds to the probability of being in stage \( i \) at time \( t \). By definition of a stationary Markov process

\[
 \pi[t+1] = \pi[t] \cdot P \tag{3.17}
\]

If \( \pi[0] \) is the initial condition vector, then

\[
 \pi[t] = \pi[0] \cdot P^t \tag{3.18}
\]
Now, since the signature register must be in one of the states after a transition,

\[ \sum_{j=1}^{2^k} P_{ij} = 1 \]  \hspace{1cm} (3.19)

Also, each state of the register has exactly two possible predecessors, then

\[ \sum_{i=1}^{2^k} P_{ij} = 1 \]  \hspace{1cm} (3.20)

For the example of Figure 3.13.c, the transition state matrix would be

\[
P = \begin{bmatrix}
1-p & p & 0 & 0 \\
0 & 0 & 1-p & p \\
0 & 0 & p & 1-p \\
p & 1-p & 0 & 0
\end{bmatrix}
\]

There exists a vector \( \pi(\alpha) \) such that

\[
\pi(\alpha) = \pi(\alpha) \cdot P \]  \hspace{1cm} (3.21)

and since

\[ \sum_{i=1}^{2^f} \pi_i = 1 \]  \hspace{1cm} (3.22)
then the probability of being in any of the $2^r$ states (for an r-stages LFSR) is $1/2^r$ irrespective of the initial condition. One of the $2^r$ states is the *all-zero* state. In this process, the *all-zero* state corresponds that process either remained in the *all-zero* state all the time (no aliasing $\rightarrow$ fault-free) or left that state and returned to it (aliasing). Thus the probability of aliasing is

$$P_{al} = P(0\rightarrow 0) - (1-p)^n$$

$$P_{al} = 1/2^r - (1-p)^n$$

which, for large $n$ is

$$P_{al} \approx 1/2^r.$$  \hspace{1cm} (3.23)

The probability of aliasing obtained in equation (3.23) is the same as equation (3.15) which considers equally likely error sequences.

### 3.4 ALIASING BOUNDS

Equation (3.23), gives only the steady state value of the aliasing probability. It is also important to know how this value is reached. To solve the dynamic properties of Markov process, the z-transform is used.

The z-transform of a function $f(n)$ is given as
\( Z(f(n)) = F(z) = \sum_{n=0}^{\infty} f(n) z^{-n} \)

which is similar to the Laplace transform, and it is unique.

Applying the z-transform to equation

\[ \pi(n+1) = \pi(n) P \] (3.24)

results

\[ z(\pi(z) - \pi(0)) = \pi(z) P \] (3.25)

Solving for \( \pi(z) \)

\[ \pi(z) = \pi(0) \cdot z \cdot (z \cdot I - P)^{-1} \] (3.26)

The inverse z-transform results

\[ \pi(n) = \pi(0) \cdot P^{-1} \{ z \cdot (z \cdot I - P)^{-1} \} \] (3.27)

or

\[ \pi(n) = \pi(0) \cdot P^n \] (3.28)
The dynamic properties of the Markov process can be studied by the analysis of $F^{-1}(z(I-P)^{-1})$.

To invert the matrix $(zI-P)$, it is necessary to calculate its determinant, which is the denominator of every term of the inverted matrix. The location of the poles (zeros of the determinant) controls the dynamic properties of the Markov process.

From the z-transform properties, poles can be anywhere in the complex $z$ plane. Poles outside the unit circle (resulting exponentially increasingly sinusoids) do not occur in the Markov process, since the probability of aliasing approaches $2^{-r}$ (equation 3.24). Complex pairs inside the circle correspond to exponentially damped sinusoids.

This Markov process is an ergodic process, which means that any state can be reached from any state and the system reaches a steady-state condition, regardless of the initial condition. Then, for any ergodic process, the determinant always has a factor $(z-1)$, which results in a steady-state response (the final value of $P_a = 2^r$).

Additionally, the smaller the value of $p$, the smaller will be the amplitude of the sinusoids. For $p$ close to zero, the sinusoid amplitude is almost zero, and for $p$ close to 1, it reaches its highest value. For $p<0.5$, the probability of aliasing asymptotically approaches $2^r$ (without going over the $2^r$ limit). On the other hand, for $p>0.5$, the probability of aliasing oscillates above and bellow the $2^r$ limit, and it stabilizes at the $2^r$ limit.
The aliasing probability (as a function of the test length) corresponding to the example of Figure 3.13.c is presented in Figure 3.14. The graph shows the probability of aliasing for four different values of $p$.

![Figure 3.14 - Aliasing Probability for $f(x) = x^3 + x^2 + 1$ as Function of Length.](image)

The graph of Figure 3.14 is obtained from [Williams 88], which also introduces the concepts of z-transform in the Markov process to study the LFSR.
Figure 3.15 presents another aliasing probability as function of testing length, for $x^3 + x + 1$. The graph shows the probability of aliasing for four different values of $p$.

![Graph showing aliasing probability for $f(x) = x^3 + x + 1$ as function of length.]

- $p=0.1$ - $p=0.2$ - $p=0.8$ - $p=0.9$

Figure 3.15 - Aliasing Probability for $f(x) = x^3 + x + 1$ as Function of Length.

The aliasing probability of the LFSR as function of the test length for $p$ below and above 0.5 is confirmed through extensive simulations performed by [Williams 86], [Ivanov 88], [Williams 88], [Ivanov 89], [Damiani 89] and [Ivanov 91], among others.
4. PSEUDO-EXHAUSTIVE TESTING

Exhaustive testing guarantees that all single stuck-at faults (referred to here only as stuck-at faults) are detected, but it is not practical for most VLSI circuits, because of the excessive testing time required.

Some techniques called pseudo-exhaustive testing have been proposed to reduce the testing time of the exhaustive testing, while keeping its advantages. These pseudo-exhaustive techniques apply exhaustive testing to portions of the CUT rather than the entire circuit.

As a simple example, consider a circuit with \( n \) inputs, which would require \( 2^n \) test patterns for exhaustive testing. If such circuit could be divided into two subcircuits, each one with \( n/2 \) inputs, the exhaustive testing time of each sub-circuit would require \( 2^{n/2} \) test patterns, which is much smaller than \( 2^n \) (specially for large \( n \)).

The basic pseudo-exhaustive testing techniques are partitioning, logic sensitization, logic verification and segmentation.

4.1 PARTITIONING

Partitioning in this work refers to hardware partitioning of a circuit into two or more subcircuits. Each sub-circuit can be tested if its inputs can be controlled and their its observed.
Consider circuit C of Figure 4.1.a, with input X and output Y, which can be decomposed into subcircuits \( C_1 \) and \( C_2 \). Sub-circuit \( C_1 \) has inputs \( X \) (primary inputs) and \( B_{21} \) (from sub-circuit \( C_2 \)). Sub-circuit \( C_1 \) has outputs \( Y \) (primary output) and \( B_{12} \) (input to sub-circuit \( C_2 \)). Sub-circuit \( C_2 \) has inputs \( X_2 \) (primary inputs) and \( B_{12} \) (from sub-circuit \( C_1 \)). Sub-circuit \( C_2 \) has outputs \( Y_2 \) (primary output) and \( B_{21} \) (input to sub-circuit \( C_1 \)).

![Diagram of circuit partitioning](image)

**Figure 4.1 - Basic Circuit Partitioning.** (a) Original, (b) Partitioned.

Sub-circuit \( C_1 \) could be tested if \( B_{21} \) could be controlled and if \( B_{12} \) could be observed. One possible approach is given if Figure 4.2.a, which requires additional input \( B_x \) and output \( B_y \), with the input \( B_{21} \) is taken through the multiplexer from \( B_x \).

A better approach is presented in Figure 4.2.b, in which \( B_{21} \) is taken from \( X_2 \) and \( B_{12} \) is observed at \( Y_2 \) through the multiplexer.
Testing of sub-circuit $C_2$ also would require similar circuitry. The configuration of Figure 4.3 can be used to perform the testing of either $C_1$ or $C_2$. Observe that in the normal mode, each sub-circuit can operate as in the original circuit. In the test mode, during test of $C_1$, $C_2$ is completely disabled (inputs and outputs), $C_1$ receives inputs from $X_1$ and $B_{21}$ (via $X_2$), and the test results are presented on $Y_1$ and $B_{21}$ (via $Y_2$).
This configuration would be practical only if the number of lines of input X (original circuit) is larger than the number of lines acting as input of each sub-circuit (primary and multiplexed). If not, testing of the partitioned circuit takes longer than the original circuit.

As an example, consider the ALU/Function Generator SN54/74LS181 shown in Figure 4.4, where the dotted line indicates subcircuits for a suggested partitioning. Figure 4.5.a shows the multiplexer partitioning of this circuit, in which, for simplicity, just the block diagrams are presented.
Exhaustive testing of the $N_1$ block requires $32 \ (2^5)$ test patterns, and testing of block $N_2$ requires $1024 \ (2^{10})$ test patterns. Pseudo-exhaustive testing of the SN54/74LS181 requires 1056 test patterns, against 16384 ($2^{14}$) for the exhaustive testing.
Unfortunately, this procedure requires additional overhead (multiplexer, wiring, etc) and few circuits could be partitioned. Also, the addition of multiplexers may reduce the speed of the circuit.

4.2 LOGIC SENSITIZATION

Paths from inputs to a sub-circuit and from the sub-circuit to an output can be sensitized, which can be used to exhaustively test the sub-circuit. This approach, called logic sensitization, does not require additional circuitry, but it requires logic analysis of the CUT.

Consider the ALU/Function Generator SN54/74LS181 of Figure 4.4. Taking $M=1$ and $S_2S_3=00$, lines $H_i$ ($0 \leq i \leq 3$) are set to 1, and the lines $L_4$ will appear at the
corresponding $F_i$ output lines. This can be seen in the block diagram of Figure 4.6.a. The lines $A_i$ are connected together, as are the lines $B_i$. Now $A, B, S_0$ and $S_1$ can be exhaustively tested with 16 ($2^4$) test patterns.

Setting $M=1$ and $S_0S_1=00$, lines $L_i$ are set to 1, and lines $H_i$ are present at the respective outputs $F_p$ as shown in Figure 4.6.b. This way, $A, B, S_2$ and $S_3$ can be exhaustively tested (also with 16 test patterns). These two procedures test the $N_i$ circuits. Sub-circuit $N_2$ can be tested by setting $S_0S_1S_2S_3=0001$ and exhaustively applying tests to lines $A_p, B_p, M$ and $C_n$, which corresponds to 1024 ($2^{10}$) test patterns.

Using logic sensitization, then the SN54/74LS181 can be pseudo-exhaustively tested with 1056 ($1024+16+16$) test patterns, compared to 16384 ($2^{14}$) of the exhaustive testing.

Figure 4.6 - Sensitized Testing. (a) $L_i$ Functions, (b) $H_i$ Functions.
As another example, consider the two-stage adder of Figure 4.7. Setting $A_4B_4=00$, lines $A_o-A_3$ and $B_o-B_3$ can be exhaustively tested with $256 (2^8)$ test patterns. The output lines to be observed are $S_0-S_3$, and $S_4$ (to observe the carry-out of the first stage, since $A_4$ and $B_4$ are 0). The second stage can be tested by setting $A_3B_3=00$ (which causes the carry-out of the first stage to be 0) and exhaustively applying test patterns to lines $A_4-A_7$ and $B_4-B_7$ (also 256 test patterns). Again, the second stage should be exhaustively tested, but this time with $A_3B_3=11$ (which causes the carry-out of the first stage to be 1), resulting another 256 test patterns. The two stages adder is pseudo-exhaustively tested with 768 ($256+256+256$) test vectors, against the 65536 ($2^{16}$) of exhaustive testing.

![Two-Stage Adder diagram](image)

Figure 4.7 - Two-Stage Adder.

Although this method does not require additional circuitry, it is time consuming in terms of circuit analysis to find "good" subcircuits and their respective
path sensitizations. Another problem is that not all circuits allow this type of pseudo-exhaustive approach.

4.3 VERIFICATION TESTING

Most combinational circuits have more than one output and in many cases each output does not depend on all inputs. Consider, as an example, the simple circuit of Figure 4.8. Each output depends on three inputs. It can be verified that output \( r \) depends on \( v, x \) and \( y \), and it can be tested with only 8 \((2^3)\) test patterns, which is the same for output \( s \) that depends on \( x, y \) and \( z \).

![Simple Circuit for Verification Testing](image)

Figure 4.8 - Simple Circuit for Verification Testing.

Analysis of the input dependence of each output is conveniently represented in the Dependence Matrix, where each row represents an output and each column represents an input. An entry is 1 if the corresponding output depends on the
corresponding input. The Dependence Matrix of the simple circuit of Figure 4.8 is given in Table 4.1.

<table>
<thead>
<tr>
<th></th>
<th>v</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.1 - Dependence Matrix of circuit of Figure 4.8.

The maximum number of inputs that an output depends on is the maximum output weight, or \( w \). The output weight corresponds to the number of ONEs on each row of the dependence matrix, and the maximum output weight corresponds to the row of larger output weight. The maximum output weight for the example of Table 4.1 is 3.

A CUT with \( w < n \) can be tested basically in two ways: using LFSR with \( f(x) \) based on Linear Codes or through rewiring of the CUT inputs during test. The latter requires additional rewiring, but it requires a smaller number of latches. The former requires one latch for each CUT input, but it has a longer testing time. These will be discussed after the automatic dependence matrix generation.
4.3.1 AUTOMATIC GENERATION OF DEPENDENCE MATRIX

A program called CONVERT has been developed (in C language) to obtain the dependence matrix of a given circuit. The program is listed in Appendix A2 and its user's guide in Appendix A1. The program has been developed as part of this work, with the objective of obtaining automatic dependence matrices for large circuits, where the manual task is virtually impossible.

The program takes as input the circuit description at gate level, in the ISCAS85 benchmark format (circuit.isc). As an output the program provides the respective dependence matrix (circuit.w) and the maximum output weight. The program also obtain the circuit description converted to TESS format [Patel 87]. Appendix A3 shows the ISCAS85 benchmark C17.isc used as an example input file, and presents the respective output files C17.w (dependence matrix) and C17.tes (TESS description format).

CONVERT is composed of basically two sections. The first section makes the conversion from ISCAS85 format to TESS format, the second section generates the dependence matrix.

In the second section, the program finds the dependence matrix by tracking the path dependencies from circuit inputs to the outputs. The program assigns a BYTE to each node of the circuit, and each input is associated to a unique bit position in the BYTE. The BYTE corresponding to each input node has a 1 in its bit position and 0 for the remaining bits. The values of the BYTES of the remaining
nodes (gate outputs) correspond to all of their respective input bytes ORed together. This is a simplified program overview and it does not reflect its implementation problems.

As an example consider again Figure 4.8 (consider, for simplicity, a BYTE of being only four bits long). Based on the previous explanation of the program,

\[
\begin{align*}
BYTE_v & : 1000 \\
BYTE_x & : 0100 \\
BYTE_y & : 0010 \\
BYTE_z & : 0001 \\
BYTE_p & : BYTE_v \text{ or } BYTE_x : 1000 \text{ or } 0100 : 1100 \\
BYTE_q & : BYTE_x \text{ or } BYTE_y : 0100 \text{ or } 0010 : 0110 \\
BYTE_r & : BYTE_p \text{ or } BYTE_q : 1100 \text{ or } 0110 : 1110 \\
BYTE_s & : BYTE_q \text{ or } BYTE_z : 0110 \text{ or } 0001 : 0111 \\
\end{align*}
\]

Since \(BYTE_r\) has 1 on the three first-bit positions, it means that output \(r\) depends on inputs \(v, x\) and \(y\). In the same way, output \(s\) depends on inputs \(x, y\) and \(z\), because \(BYTE_s\) has 1 on the three last bit positions.

**4.3.2 VERIFICATION TESTING BASED ON REWIRING**

In the Verification Testing based on rewiring, the inputs of the CUT are rewired during test. Consider again the circuit given in Figure 4.8. During the test the inputs \(w\) and \(z\) can be connected together, since no output depends on both simultaneously. The new rewired circuit is now composed of three sets of inputs, \((x)\), \((y)\) and \((w,z)\), and it can be exhaustively tested with \(2^3\) test patterns.

In this method, the number of sets of inputs, \(p\), must be minimum, where \(w \leq p \leq n\). This is a limitation since each set is connected to a distinct stage of a test.
pattern generator. Unfortunately, the major difficulty of this approach is the proper selection of sets of inputs. This problem falls in the class of NP-complete problems.

An approach is proposed in [Hirose 82], but the complexity and length of such approach are so discouraging that the authors suggest a simplified version of their method. However, the simplified version does not provide satisfactory results.

As part of this work, a new method has been developed to obtain proper sets of inputs in a reasonable amount of time. The method is based on scheduling used in parallel processing [Hwang 84] and test scheduling [Chen 91]. The method adopts the terminology used in [Barzilai 81]. Two inputs \( x \) and \( y \) are called adjacent if there exists at least one circuit output that depends on the both inputs \( x \) and \( y \). Two inputs lines can be connected together (during test) only if they are non-adjacent.

A non-adjacency graph can be created to represent the dependence matrix [Barzilai 81]. In this graph, each node represents an input, and an arc is placed between two nodes \( i \) and \( j \) if and only if the pair \( (i, j) \) in non-adjacent. Consider, as an example, the circuit of Figure 4.9.a. The corresponding dependence matrix is given in Table 4.2 and the non-adjacency graph is presented in Figure 4.9.b. It can be seen that nodes 1 and 2 are not non-adjacent (or adjacent), since output \( x \) depends on both. On the other hand, nodes 1 and 4 are non-adjacent, since no output depends on them.
Figure 4.9 - Merging Example. (a) Circuit, (b) Non-Adjacency Graph.

Table 4.2 - Dependence matrix of circuit of Figure 4.9.a.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>w</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The objective of the procedure is to merge as many nodes as possible, so that the minimum number of sets of inputs is obtained. Merging nodes corresponds to selects inputs that are placed together during test. Nodes are merged based on the graph topology.
In this method, node $k$ is a common neighbor of the pair $(i,j)$ if there are arcs from $k$ to both nodes $i$ and $j$. For instance, in Figure 4.9.b, node 3 is a common neighbor of nodes 1 and 4. Merging nodes $i$ and $j$ will cause the deletion of few edges;

i. Edge $(i,j)$ itself, since nodes $i$ and $j$ are merged.

ii. If node $k$ is a neighbor of either node $i$ or $j$, then either existing edge $(i-k)$ or $(j-k)$ is deleted.

iii. If node $k$ is a common neighbor, either edge $(i-k)$ or $(j-k)$ is deleted. The weight of the remaining node is the sum of its previous weight, the weight of the deleted node, and the weight of the merged edge $(i-j)$. Where the initial weight of an edge $(i-j)$ is the average of the number of edges arriving at nodes $i$ and $j$.

A group of guidelines has been obtained to select the nodes to be merged. These guidelines are based on the circuit topology. The guidelines are presented in the form of priorities for edge deletion (node merging);

i. **Minimum number of deleted edges**, the merged nodes that cause a minimum number of deleted nodes is selected.

ii. **Maximum number of common neighbors**, the merged pair that has a maximum number of common neighbors.

iii. **Edge with higher weight**, the merged node with higher weight. If more than one edge is found, either edge can be deleted.
Initially, a table is created containing the candidate edges to be merged. The table contains the number of deleted nodes for each merged node, and their respective weights. After each merging procedure, the table must be updated. The merging procedure is repeated until the table is exhausted.

Consider again the circuit of Figure 4.9.a and its respective non-adjacency graph of Figure 4.9.b. The initial merging table obtained is presented in Table 4.3.

<table>
<thead>
<tr>
<th>merged edges</th>
<th>deleted edges</th>
<th>common neighbor</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-3</td>
<td>4</td>
<td>2</td>
<td>3.5</td>
</tr>
<tr>
<td>1-4</td>
<td>5</td>
<td>1</td>
<td>3.5</td>
</tr>
<tr>
<td>1-5</td>
<td>5</td>
<td>1</td>
<td>3.5</td>
</tr>
<tr>
<td>1-6</td>
<td>5</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2-4</td>
<td>5</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2-5</td>
<td>5</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2-6</td>
<td>4</td>
<td>0</td>
<td>2.5</td>
</tr>
<tr>
<td>3-4</td>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3-5</td>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

As can be observed in Table 4.3, nodes 1 and 3 are candidates to be merged. They have 2 common neighbors, nodes 4 and 5. The weight is 3.5, average of 4 arriving at mode 1 and 3 arriving at node 3. If nodes 1 and 3 are merged, 4 edges are deleted; edge (1-3) itself, plus edge (1-6) since node 6 is not a common neighbor. Node 5 is a common neighbor, so either (1-5) or (4-5) is deleted. Additionally, either
edges (1-4) or (3-4) is deleted since node 4 is a common neighbor. The remaining entries of the Table 4.3 are similarly obtained.

The first deletion priority selects (1-3), (3-4) and (3-5). They are narrowed to (1-3) by the second deletion priority. The new non-adjacency graph obtained is given in Figure 4.10.a, and the new corresponding deletion table is given as Table 4.4.

\[\text{Figure 4.10 - Non-Adjacency Graph. (a) After First Merging, (b) After Second Merging.}\]

<table>
<thead>
<tr>
<th>merged edges</th>
<th>deleted edges</th>
<th>common neighbor</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-4</td>
<td>3</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1-5</td>
<td>3</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>2-4</td>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2-5</td>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2-6</td>
<td>3</td>
<td>0</td>
<td>2.5</td>
</tr>
</tbody>
</table>
For Table 4-4, the first deletion priority selects (1-4), (1-5) and (2-6), which is kept by the second deletion priority. The third deletion priority selects (1-4) and (1-5), and now either one could be selected. Supposing (1-4) is selected, the new obtained non-adjacency graph is shown in Figure 4.10.b, and the respective deletion table is given as Table 4.5.

Table 4.5 - Candidates after second selection.

<table>
<thead>
<tr>
<th>merged edges</th>
<th>deleted edges</th>
<th>common neighbor</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-5</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2-6</td>
<td>2</td>
<td>0</td>
<td>2.5</td>
</tr>
</tbody>
</table>

The first and second deletion priorities select (2-5) and (2-6), which is narrowed to (2-5) by the last deletion priority.

The relation of merged nodes obtained is (1-3), (1-4) and (2-5), which means that nodes 1, 3 and 4 are placed in the same set of inputs. By the same way nodes 2 and 5 are also placed in the same set, but node 6 remains alone in a set. Three primary sets of input have been obtained, namely (1-3-4), (2-5) and (6). Adopting this set of inputs, the pseudo-exhaustive testing requires 8 \(2^3\) test patterns, opposed to 64 \(2^6\) of the exhaustive testing.

This procedure has been implemented in C and the listing of the program (LAST) is given in Appendix B2, and the user’s guide in Appendix B1. An example of input and output files is given in Appendix B3.
4.3.3 VERIFICATION TESTING BASED ON LINEAR CODES

A circuit with \( n \) inputs and maximum output weight \( w \), represented as \((n,w)\)CUT, can be tested by a \((n,k)\)LFSR. The theory of Verification Testing using Linear Codes is based on the work of [Wang 85b] and [Wang 86a].

As presented in Chapter Two, a given \((n,k)\)LFSR with characteristic polynomial \( f(x) = g(x) \times p(x) \), and initial state \( S_0(x) \), divisible by \( g(x) \), will generate \( 2^k-1 \) patterns \( (g(x) \) of degree \( n-k \) and \( p(x) \) of degree \( k \)). Theorem 4.1 may be used to evaluate the value of \( k \) as a function of \( w \).

**Theorem 4.1**

An \((n,k)\)LFSR is capable of pseudo-exhaustively test an \((n,w)\)CUT if \( k \) is the smallest integer such that

\[
w = \left\lfloor \frac{k}{n-k+1} \right\rfloor + \left\lceil \frac{k}{n-k+1} \right\rceil \tag{4.1}\]

where \( [\alpha] \) is the greatest integer less than or equal to \( \alpha \), and \( [\alpha] \) is the smallest integer larger than or equal to \( \alpha \). The proof given in Appendix C.
The general procedure for the design of an \((n,k)\)LFSR for the test of an \((n,w)\)CUT can be given in the form of an algorithm, as follows:

i. obtain \(w\) (using CONVERT);

ii. compute \(k\) (using equation 4.1);

iii. find a primitive polynomial \(p(x)\) of degree \(k\) [Bardell 87];

iv. find a polynomial of degree \(n-k\);

v. compute \(f(x) = g(x) \ast p(x)\);

vi. take the seed \(S_0(x)\) as any multiple of \(g(x)\). This could be \(g(x)\) itself, for simplicity.

The \((n,k)\)LFSR thus obtained will have period \(T = 2^k - 1\), and will be able to pseudo-exhaustively test the \((n,w)\)CUT. As an example, consider testing a \((10,5)\)CUT. Based on relation 4.1, a \((10,8)\)LFSR should be used. From [Bardell 87] or [Golomb 82], \(p(x) = 1 + x + x^5 + x^6 + x^8\), and \(g(x) = 1 + x + x^2\) (of degrees 8 and 2, respectively), obtaining \(f(x) = 1 + x^3 + x^5 + x^9 + x^{10}\). For simplicity, the seed could be taken as \(S_0(x) = g(x) = 1 + x + x^2\).

4.4 SEGMENTATION

If the previous pseudo-exhaustive techniques are not suitable or do not provide reasonable reduction in testing time, the segmentation method might be employed. Segmentation consists of dividing a large circuit into smaller subcircuits.
The purpose of segmentation is to limit the maximum weight of any node, that is, to limit the maximum number of inputs that an output (or any node) depends on. Limiting the maximum weight of an output node will reduce the testing time of the circuit.

Segmentation can be achieved by insertion of latches (flip-flops) that could act as BILBO. It also can be achieved by using additional logic gates that allow observation and control on the segmented points. Consider a typical section of a circuit given in Figure 4.11.a, where the output of gate $P$ is fed into gate $Q$. This circuit could be partitioned as in Figure 4.11.b, where the additional circuitry is inserted between gates $P$ and $Q$. Note that the output of gate $P$ can be directly observed, and (when test line is at logic level 1), the value at input of gate $Q$ can be controlled. Normal operation of the circuit occurs when test line is at logic level 0.

To simplify the circuit analysis, the circuit $C$ will be represented by a direct acyclic graph $G_c$. In this graph representation, squares represent inputs and outputs, and circles represent logic gates, or macros, such as flip-flops or "unbreakable" circuit sections. Directed edges correspond to signal flow of the corresponding gates of the circuit $C$. The points where the partition is going to take place are represented by black squares, which will be referred to as pseudo-inputs and pseudo-outputs.
The graph corresponding to the circuit of Figure 4.11.a is shown in Figure 4.11.c.

The objective is to partition the acyclic graph $G_c$ into disjoint segments by adding the black squares (or interconnections). In each segment, each circle should not depend on more than $\beta$ inputs, and the number of segments should be kept at minimum. The number of black squares must be kept at a minimum, since those represent additional circuitry. This problem has been proved to be NP complete [Jone 88].
For most circuits, the value of $\beta$ is 20 [Jone 88], [Shperling 87], [Bhatt 86] and [Archambeau 85]. This value of $\beta$ is a good compromise between testing time, circuit size and amount of added circuitry.

A simplistic approach is proposed by [Roberts 84]. In this approach number of contributors refers to the number of squares that circle depends on. Starting at a square (input), a route is taken to an output. At each advance, the number of contributors is checked to ensure that it is smaller than $\beta$. If $\beta$ is not reached, the next circle is considered.

As an example, consider Figure 4.12.a which shows a graph to be partitioned. For simplicity, consider $\beta$ equals to 3. Figure 4.12.b illustrates the respective graph segmented.

Unfortunately, each additional black square added represents additional circuitry to the original circuit. This additional circuit may cause performance degradation, especially if the critical path of the original circuit is reached. Although this approach can be seen as pseudo-exhaustive testing, it should considered as a last resort. The other pseudo-exhaustive testing methods, namely, partitioning, logic sensitization and logic verification should be used when possible.
Figure 4.12 - Segmentation. (a) Original Graph, (b) Segmented Graph.

Pseudo-exhaustive testing can provide shorter testing time than exhaustive testing without jeopardizing the fault coverage, as discussed in this Chapter. Along with a methodology for the automatic generation of the dependence matrix, this Chapter also presented the methodology developed to obtain the proper sets of inputs for verification testing. The next Chapter presents the Reduced Test Methods.
5. REDUCED TEST METHODS

Since testing accounts for nearly a third of VLSI costs [Levitt 92], any reduction in testing time represents a reduction on the overall VLSI cost. The pseudo-exhaustive testing methods presented in Chapter Four (specifically the verification method of section 4.3) allow reduction of testing time when compared to full exhaustive testing. Even with this reduction, the testing time still can long, thus suggesting further reductions.

The reduced test methods proposed in this chapter are based on existing pseudo-exhaustive testing methods, thus causing no additional costs in terms of circuitry when compared to pseudo-exhaustive testing methods.

In the reduced test methods, only a fraction of the pseudo-exhaustive testing test set is applied to the CUT, without reducing the fault coverage.

5.1 REDUCED TEST METHOD BASED ON LINEAR CODES

In the Reduced Rest Method based on Linear Codes, instead of applying all $2^k$ test patterns to the (n,w)CUT from a (n,k)LFSR, fault simulation is used to identify the minimum number of consecutive test patterns that would detect all faults.

In this method an (n,k)LFSR is designed according to the procedure presented in Chapter Four. Each vector generated by the (n,k)LFSR is used as a test pattern in the fault simulation of the (n,w)CUT.
In this process, for each test pattern generated by the LFSR, the CUT is fault simulated and the faults detected are marked. If not all faults have been marked, the LFSR generates another test pattern. This process continues until all faults have been marked. Note that the number of test vectors generated by the \((n,k)\)LFSR during simulation corresponds to the number of clock cycles required in the real circuit to detect all faults.

The \((n,w)\)CUT can be simulated again for another \((n,k)\)LFSR, implementing a different \(f(x)\) (either changing \(p(x)\), \(g(x)\) or both). Among those simulated, the \((n,k)\)LFSR resulting in the smallest number of test patterns (able to detect all faults) is selected to be implemented in the actual circuit. Once again, it should be noted that the number of test vectors generated by the \((n,k)\)LFSR during simulation corresponds to the number of clock cycles required in the real circuit to detect all faults.

The general procedure for the design of the an \((n,k)\)LFSR to test an \((n,w)\)CUT implementing the \textit{Reduced Test Method based on Linear Codes} is presented in flowchart of Figure 5.1.

The flowchart of Figure 5.1 can be presented in the form of an algorithm, given in the Algorithm 5.1.
ANALYZE CUT TO OBTAIN W (maximum output weight)

COMPUTE k (equation 4.1)

SELECT p(x) OF DEGREE k

SELECT A g(x) OF DEGREE n-k

f(x) = g(x)*p(x)

SET INITIAL STATE OF LFSR AS A MULTIPLE OF g(x)

FAULT SIMULATE THE CUT (mark all detected faults)

APPLY ONE CLOCK CYCLE TO THE LFSR

ALL FAULTS DETECTED?

Y

ANOTHER g(x)?

Y

ANOTHER p(x)?

Y

IMPLEMENT THE LFSR WITH f(x) RESULTING IN THE LOWEST NUMBER OF CLOCK CYCLES AND TAKE So(x) = g(x)

Figure 5.1 - Flow Graph to Design an (n,k)LFSR Implementing the Reduced Test Method based on Linear Codes.
Algorithm 5.1

i. Analyze the CUT to obtain the maximum output weight $w$ (using CONVERT);

ii. Compute the value of $k$ (using equation 4.1);

iii. Select a primitive polynomial $p(x)$ of degree $k$ [Bardell 87][Golomb 82];

iv. Select a polynomial $g(x)$ of degree $n-k$;

v. Compute $f(x)=f(x)*g(x)$;

vi. Establish the connections of the $(n,k)$LFSR according to $f(x)$, and set its initial state, $S_0(x)$ to any multiple of $g(x)$, it could be $g(x)$ itself;

vii. Fault simulate the CUT and mark the faults detected;

viii. Obtain a new test pattern (apply a clock cycle to the LFSR);

ix. Repeat steps vii and viii until all faults have been marked, and record the total number of clock cycles;

x. If an acceptable test length have not being obtained, pick another $g(x)$ and repeat steps iv to ix;

xi. If an acceptable test length has not being obtained, pick another $p(x)$ and repeat steps iii to x;

xii. Implement the LFSR with $f(x)$ that results in the least number of clock cycles.
Observe that there is no limitation on the procedure (graph or algorithm) regarding the numbers of possible p(x) and g(x) considered for simulation. A limitation could be the possible number of primitive polynomials p(x), given by expression (2.8), and the number of possible g(x), given by $2^{n-k}$. Another possible limitation would be the number of possible initial seeds $S_0(x)$. The major limitation, although, is the time limit that can be dedicated to fault simulation of the (n,w)CUT.

As an example of the Reduced Test Method based on Linear Codes, consider the circuit of Figure 5.2. The procedure will be demonstrated step by step.

Figure 5.2 - Test Circuit to Illustrate the Reduced Test Method based on Linear Codes.
The dependence matrix of the circuit of Figure 5.2 is given in Table 5.1. The dependence matrix was obtained using CONVERT (section 4.3 and Appendix A). It can be verified from Table 5.1 (and directly from CONVERT) that the maximum output weight $w$ is 6. Observe that the circuit has 13 inputs, thus it is a (13,6)CUT.

Table 5.1 - Dependence Matrix of the Circuit of Figure 5.2.

<table>
<thead>
<tr>
<th>Output</th>
<th>in0</th>
<th>in1</th>
<th>in2</th>
<th>in3</th>
<th>in4</th>
<th>in5</th>
<th>in6</th>
<th>in7</th>
<th>in8</th>
<th>in9</th>
<th>in10</th>
<th>in11</th>
<th>in12</th>
</tr>
</thead>
<tbody>
<tr>
<td>out1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>out2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>out3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>out4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>out5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>out6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>out7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>out8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

From relation (4.1), the value of $k$ is 11. A primitive polynomial of degree 11 could be $p(x) = 1 + x^2 + x^{11}$ and $g(x)$ can be any polynomial of degree 2. For each $g(x)$, indicated in Table 5.2, the product $f(x) = g(x) \cdot p(x)$ is obtained, and the initial value $S_0(x)$ is set equal to $g(x)$.

The (13,6)CUT is fault simulated for each (13,11)LFSR, and the respective number of test patterns that detect all faults is recorded in Table 5.2. The procedure is repeated for another primitive polynomial $p(x)$ and the respective number of test patterns is also recorded in Table 5.2.
Table 5.2 - Number of Test Patterns (Circuit of Figure 5.2).

<table>
<thead>
<tr>
<th>g(x)</th>
<th>(p_1(x) = 1 + x^2 + x^{11})</th>
<th>(p_2(x) = 1 + x^3 + x^{11})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x^2 + x + 1)</td>
<td>139</td>
<td>101</td>
</tr>
<tr>
<td>(x^2 + x)</td>
<td>133</td>
<td>74</td>
</tr>
<tr>
<td>(x^2 + 1)</td>
<td>115</td>
<td>84</td>
</tr>
<tr>
<td>(x^2)</td>
<td>136</td>
<td>110</td>
</tr>
</tbody>
</table>

It can be observed that pseudo-exhaustive testing of the circuit of Figure 5.2 requires 2048 (2\(^{11}\)) test patterns or clock cycles, while using the Reduced Test Method based on Linear Codes, it requires only 74 clock cycles to detect all faults. The Reduced Test Method based on Linear Codes takes, for this case, only 3.6% of the pseudo-exhaustive testing time. If the search is extended to other primitive polynomials, or to other initial seeds \(S_0(x)\), shorter testing time may be obtained.

The Reduced Test Method based on Linear Codes has been applied to the TI SN54/74LS630 (error detection and correction circuit). The TI SN54/74LS630 has 23 inputs and a maximum output weight of 10, and the value obtained for k is 20. Table 5.3 presents the results if \(p(x) = 1 + x^3 + x^{20}\) is used. It can be verified that the test length is 20 when the Reduced Test Method based on Linear Codes is used, compared to \(2^{20}\) of the pseudo-exhaustive testing (less than 1%).
Table 5.3 - Reduced Test Method based on Linear Codes
Applied to TI SN54/74LS630 for \( p(x) = 1 + x^3 + x^{20} \).

<table>
<thead>
<tr>
<th>( g(x) )</th>
<th>clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x^3 + x^2 + x + 1 )</td>
<td>20</td>
</tr>
<tr>
<td>( x^3 + 1 )</td>
<td>20</td>
</tr>
<tr>
<td>( x^3 + x + 1 )</td>
<td>20</td>
</tr>
<tr>
<td>( x^3 + x^2 + 1 )</td>
<td>20</td>
</tr>
<tr>
<td>( x^3 + x^2 + x )</td>
<td>21</td>
</tr>
<tr>
<td>( x^3 + x )</td>
<td>21</td>
</tr>
<tr>
<td>( x^3 + x^3 )</td>
<td>22</td>
</tr>
<tr>
<td>( x^3 )</td>
<td>23</td>
</tr>
</tbody>
</table>

The procedure has been applied to some ISCAS85 benchmark circuits [Brglez 85]. The results of the method being applied to the c880.isc is shown in Table 5.4. The c880.isc has 60 inputs and the maximum output weight is 45, where \( k \) is found to be 59. For the \( p(x) \) and \( g(x) \) used, the best result is 5101, which corresponds to less than 1% of the \( 2^{59} \) test vectors of the pseudo-exhaustive testing.

Table 5.4 - Reduced Test Method based on Linear Codes Applied to c880.isc.

<table>
<thead>
<tr>
<th>( p(x) )</th>
<th>( g(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x + 1 )</td>
<td>( x )</td>
</tr>
<tr>
<td>( x^{59} + x^{22} + x^{21} + x + 1 )</td>
<td>35061</td>
</tr>
<tr>
<td>( x^{59} + x^{58} + x^{18} + x^{37} + 1 )</td>
<td>5101</td>
</tr>
<tr>
<td>( x^{59} + x^{5} + x^{5} + x^{4} + x^{3} + x + 1 )</td>
<td>8211</td>
</tr>
<tr>
<td>( x^{59} + x^{58} + x^{54} + x^{55} + x^{54} + x^{43} + 1 )</td>
<td>20139</td>
</tr>
</tbody>
</table>
The results of the procedure when applied to the benchmark c5315.isc are presented in Table 5.5. The c5315.isc has 178 inputs, with maximum output weight 67, and resulting k of 174. Note that, again, the test length using this method represents less than 1% of the test length of the pseudo-exhaustive testing. The best option for this table (2208 clock cycles) is \( p(x) = x^{174} + x^{161} + 1 \), and \( g(x) = S_0(x) = x^4 + x^3 + x^2 \), forming \( f(x) = x^{178} + x^{177} + x^{176} + x^{165} + x^{164} + x^{163} + x^4 + x^3 + x^2 \).

Table 5.5 - Reduced Test Method based on Linear Codes Applied to c5315.isc.

<table>
<thead>
<tr>
<th>( g(x) )</th>
<th>( p(x) )</th>
<th>( 1 + x^{13} + x^{174} )</th>
<th>( 1 + x^{141} + x^{174} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x^4 + x^3 + x^2 + x + 1 )</td>
<td>21888</td>
<td>5536</td>
<td></td>
</tr>
<tr>
<td>( x^4 + x^3 + x^2 + x )</td>
<td>4896</td>
<td>4096</td>
<td></td>
</tr>
<tr>
<td>( x^4 + x^3 + x^2 + 1 )</td>
<td>20832</td>
<td>8768</td>
<td></td>
</tr>
<tr>
<td>( x^4 + x^3 + x + 1 )</td>
<td>29920</td>
<td>9024</td>
<td></td>
</tr>
<tr>
<td>( x^4 + x^2 + x + 1 )</td>
<td>20128</td>
<td>7904</td>
<td></td>
</tr>
<tr>
<td>( x^4 + x^3 + x^2 )</td>
<td>5568</td>
<td>2208</td>
<td></td>
</tr>
<tr>
<td>( x^4 + x^3 + x )</td>
<td>2976</td>
<td>3968</td>
<td></td>
</tr>
<tr>
<td>( x^4 + x^2 + x )</td>
<td>3936</td>
<td>2880</td>
<td></td>
</tr>
<tr>
<td>( x^4 + x^3 + 1 )</td>
<td>25856</td>
<td>6432</td>
<td></td>
</tr>
<tr>
<td>( x^4 + x^2 + 1 )</td>
<td>31104</td>
<td>4032</td>
<td></td>
</tr>
<tr>
<td>( x^4 + x + 1 )</td>
<td>37120</td>
<td>11904</td>
<td></td>
</tr>
<tr>
<td>( x^4 + 1 )</td>
<td>62976</td>
<td>8768</td>
<td></td>
</tr>
<tr>
<td>( x^4 + x )</td>
<td>5920</td>
<td>4672</td>
<td></td>
</tr>
<tr>
<td>( x^4 + x^2 )</td>
<td>4672</td>
<td>2912</td>
<td></td>
</tr>
<tr>
<td>( x^4 + x^3 )</td>
<td>3328</td>
<td>4096</td>
<td></td>
</tr>
<tr>
<td>( x^4 )</td>
<td>21664</td>
<td>6656</td>
<td></td>
</tr>
</tbody>
</table>
Table 5.6 shows the percentage of the faults detected for up to 1,000,000 test patterns. The c7552.isc has 207 inputs (n), maximum output weight (w) 104, and the obtained k is 206. The fsimm.ou can not obtain 100% fault coverage for this case since it is unable to fault simulate more than 1,000,000 test patterns.

Table 5.6 - Reduced Test Method Applied to c7552.isc
(Percentage of Faults Detected up to 1,000,000 Test Patterns).

<table>
<thead>
<tr>
<th>( p(x) )</th>
<th>( g(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x^{206} + x^{147} + x^{2} + x + 1 )</td>
<td>99.81 99.87</td>
</tr>
<tr>
<td>( x^{206} + x^{205} + x^{104} + x^{19} + 1 )</td>
<td>99.66 99.82</td>
</tr>
</tbody>
</table>

5.2 SIMULATION RESOURCES

Fault simulation of the CUT along with the test patterns generated by the LFSR have been accomplished by the use of a modified version of FSIM [Lee 91a]. The original version of FSIM, supplied by the Virginia Polytechnic & State University performs parallel pattern simulation for combinational circuits (see Chapter One). The test patterns can be supplied by an internal random test pattern generator, or they can be read from an external input file.

It would not be practical to generate the test patterns (simulate an LFSR) and store them in a file, where the FSIM could read them, because of the large size of the files.
The FSIM has been modified to allow the simulation of the LFSR (test pattern generation) and fault simulation of the CUT simultaneously. For this purpose, a section which simulates the LFSR has been developed and adapted into FSIM.

In the modified version of FSIM, called fsimm.ou, the user is asked to supply:

i. the number of undetectable faults;

ii. the degree of \( p(x) \);

iii. \( p(x) \), exponent of those factors whose coefficients are different of zero;

iv. \( g(x) \), exponent of those factors whose coefficients are different of zero;

v. the number of test patterns to be fault simulated.

Since a circuit may have undetectable faults, fsimm.ou needs the number of undetectable faults, otherwise for 100% fault coverage, there would be no end to the program execution.

This version does not ask for \( S_0(x) \) because it takes the seed as being equal to \( g(x) \) (when \( w < n \)). If the supplied degree of \( p(x) \) is equal to the number of CUT inputs, fsimm.ou does not ask for \( g(x) \), since its degree would be zero. The program then assumes \( f(x) \) as being equal to \( p(x) \) and it asks that \( S_0(x) \) be supplied.

This versatility allows the program to simulate any other kind of LFSR with any seed, \( S_0(x) \). Such capability is important when the maximum output weight is the same as the number of inputs (\( n = w \)). This corresponds to a special case of the Reduced Test Method. Table 5.7 presents this special case applied to few ISCAS85
benchmark circuits. The table shows percentage of fault coverage for up to 1,000,000 test patterns.

Table 5.7 - Fault Coverage of ISCAS85 Benchmark Circuits for up to 1,000,000 Test Patterns.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>n</th>
<th>f(x)</th>
<th>S_{f(x)}</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1355.isc</td>
<td>41</td>
<td>x^{i1} + x^2 + 1</td>
<td>x^3 + x^2 + 1</td>
<td>79.24</td>
</tr>
<tr>
<td>c1908.isc</td>
<td>33</td>
<td>x^{i2} + x^3 + 1</td>
<td>x^4 + x^2 + 1</td>
<td>63.79</td>
</tr>
<tr>
<td>c2670.isc</td>
<td>157</td>
<td>x^{i7} + x^{i8} + x + 1</td>
<td>x^{i10} + x^2 + x^4 + x^6 + x^8 + x^9 + x^2 + x^3 + x + 1</td>
<td>54.45</td>
</tr>
<tr>
<td>c3540.isc</td>
<td>50</td>
<td>x^{i0} + x^{i7} + x^2 + x + 1</td>
<td>x^{i8} + x^4 + x^6 + x^8 + x^{i1} + x^{i9} + x^2 + x^3 + x^7 + x^8 + x^{i9} + x^{i3} + x^3 + x^7 + x^8 + x + 1</td>
<td>47.08</td>
</tr>
<tr>
<td>c432.isc</td>
<td>36</td>
<td>x^{i8} + x^{i1} + 1</td>
<td>x^{i8} + x^{i4} + x^{i3} + x^{i1} + x^{i10} + x^2 + x^3 + x^7 + x^8 + x^9 + x^2 + x^3 + x^7 + x^8 + x + 1</td>
<td>86.34</td>
</tr>
<tr>
<td>c499.isc</td>
<td>41</td>
<td>x^{i4} + x^2 + 1</td>
<td>x^4 + x^2 + 1</td>
<td>89.46</td>
</tr>
<tr>
<td>c628.isc</td>
<td>32</td>
<td>x^{i2} + x^{i7} + x + 1</td>
<td>x^{i8} + x^{i9} + x^{i3} + x^{i1} + x^{i10} + x^4 + x^8 + x^9 + x + 1</td>
<td>91.32</td>
</tr>
</tbody>
</table>

5.3 STATISTICAL REDUCTION ON THE REDUCED TEST METHOD

For large circuits, repeating the fault simulation for many different f(x) may be a time-consuming process. For such cases, instead of fault simulating the CUT for full fault coverage, the fault simulation process could be performed for only a certain percentage c of the total number of faults.

It has been shown (experimentally) [Bardell 87] and [Agrawal 88] that the number of test patterns tends to be linearly dependent on the number of faults. Each f(x) has a different dependence relationship for the faults being simulated.

Based on these assumptions, a new approach has been developed. In this approach the circuit is fault simulated only until a certain percentage c of the faults is detected. This process is repeated for many f(x) and, only for a few of those f(x) that result in short number of test patterns, full fault simulation is performed. The
best \( f(x) \) resulting from the full fault simulation is then implemented in the actual circuit.

Note that only those \( f(x) \) that result in short number of test patterns at \( c \) fault coverage are considered for the new fault simulation at full fault coverage. In the original *Reduced Test Method based on Linear Codes* all \( f(x) \) are fault simulated at full fault coverage.

This idea has been verified by the results obtained from the simulations and statistical analysis. Consider as an example the benchmark c5315.isc, whose results are shown in Table 5.8.

For any given \( f(x) \), the fault simulation resulting in 100% of the faults is placed in the first column, and the fault simulation resulting in 75% and 50% of the faults are placed in the second and third columns, respectively. It can be noted that \( f(x) \) resulting in the shortest number of test patterns at 100% fault coverage is among the four best choices at 75% fault coverage.
Table 5.8 - Statistical Reduction Applied to the c5315.isc.

<table>
<thead>
<tr>
<th>$g(x)$</th>
<th>$p(x)$</th>
<th>$1 + x^{13} + x^{174}$</th>
<th>$1 + x^{161} + x^{174}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>100%</td>
<td>75%</td>
</tr>
<tr>
<td>$x^4 + x^3 + x^2 + x + 1$</td>
<td>21888</td>
<td>1920</td>
<td>333</td>
</tr>
<tr>
<td>$x^4 + x^3 + x^2 + x$</td>
<td>4896</td>
<td>512</td>
<td>314</td>
</tr>
<tr>
<td>$x^4 + x^3 + x^2 + 1$</td>
<td>20832</td>
<td>2080</td>
<td>274</td>
</tr>
<tr>
<td>$x^4 + x^3 + x + 1$</td>
<td>29920</td>
<td>2080</td>
<td>260</td>
</tr>
<tr>
<td>$x^4 + x^2 + x + 1$</td>
<td>20128</td>
<td>1888</td>
<td>268</td>
</tr>
<tr>
<td>$x^4 + x^3 + x^2$</td>
<td>5568</td>
<td>544</td>
<td>322</td>
</tr>
<tr>
<td>$x^4 + x^3 + x$</td>
<td>2976</td>
<td>512</td>
<td>259</td>
</tr>
<tr>
<td>$x^4 + x^2 + x$</td>
<td>3936</td>
<td>512</td>
<td>268</td>
</tr>
<tr>
<td>$x^4 + x^3 + 1$</td>
<td>25856</td>
<td>2650</td>
<td>333</td>
</tr>
<tr>
<td>$x^4 + x^2 + 1$</td>
<td>31104</td>
<td>2720</td>
<td>494</td>
</tr>
<tr>
<td>$x^4 + x + 1$</td>
<td>37120</td>
<td>2752</td>
<td>442</td>
</tr>
<tr>
<td>$x^4 + 1$</td>
<td>62976</td>
<td>5120</td>
<td>655</td>
</tr>
<tr>
<td>$x^4 + x$</td>
<td>5920</td>
<td>576</td>
<td>351</td>
</tr>
<tr>
<td>$x^4 + x^2$</td>
<td>4672</td>
<td>608</td>
<td>444</td>
</tr>
<tr>
<td>$x^4 + x^3$</td>
<td>3328</td>
<td>544</td>
<td>346</td>
</tr>
<tr>
<td>$x^4$</td>
<td>21664</td>
<td>1888</td>
<td>616</td>
</tr>
</tbody>
</table>

Since the simulation time is proportional to the number of test patterns being simulated, it can be observed that the average simulation time for a percentage of the faults is much shorter than for full fault coverage. For instance, in the case of Table 5.8, simulation for 75% of the faults takes only 8.76% of the time required for full fault coverage. Also, simulation at 50% of the faults represents only 2.47% of the time required for full fault coverage.
The given approach is strengthened from the statistical analysis of the results. The correlation factor, for example of Table 5.8 is 0.98 between the 75% and 100% fault coverage. The correlation factor $r$ [Baten 38] defined as

$$r = \frac{\sum (x-x) (y-y)}{\sqrt{\sum (x-x)^2 \cdot \sum (y-y)^2}}$$

(5.1)

where $\bar{x}$ represents the average of all elements of set x. Further $-1 \leq r \leq 1$, where 1 represents the perfect positive linear relation and the -1 represents the perfect negative relation. In the perfect positive relation the values of one set of data fall in the line of projection based on the other set of data. The same analysis is true for the perfect negative relation, in which the values fall in the negative line of projection. If $r$ is zero there is no correlation between the two sets of data.

The correlation factor of 0.98 represents an almost perfect positive linear relation of the two sets of 100% and 75% fault coverage (c5315.isc). The correlation factor for the two sets 50% and 100% fault coverage is 0.68. Although it is not very high, it is still acceptable.

The data of Table 5.8 has been placed in the form of a graph, as presented in Figure 5.3. In this graph, each mark on the horizontal axis corresponds to a different $f(x)$, from Table 5.8.

This graph allows ease visual comparison of the two sets of data, where the "behavior" of the two sets of data can be observed. It can be seen that the shape of the curve at 75% and 100% fault coverage are almost identical (correlation factor
0.98). It also can be observed that at 50% and 100% fault coverage, the curves are still close to each other (correlation factor 0.68).

Figure 5.3 - Number of Test Patterns at Different Fault Coverage (c5315.isc).
Consider as another example the benchmark c880.isc, presented in Table 5.9 for 75% and 100% fault coverage simulation. Although the correlation factor seems to be low, it can be verified that the best $f(x)$ at full coverage is among the three best choices at 75%.

Table 5.9 - Statistical Reduction Applied to the c880.isc.

<table>
<thead>
<tr>
<th>$p(x)$</th>
<th>$g(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x + 1$</td>
</tr>
<tr>
<td></td>
<td>100%</td>
</tr>
<tr>
<td>$x^{59} + x^{22} + x^{21} + x + 1$</td>
<td>35061</td>
</tr>
<tr>
<td>$x^{59} + x^{58} + x^{38} + x^{37} + 1$</td>
<td>5101</td>
</tr>
<tr>
<td>$x^{59} + x^6 + x^5 + x^4 + x^3 + x + 1$</td>
<td>8211</td>
</tr>
<tr>
<td>$x^{59} + x^{58} + x^{56} + x^{55} + x^{43} + 1$</td>
<td>20139</td>
</tr>
</tbody>
</table>

The graph corresponding to the data of Table 5.9 is presented in the graph of Figure 5.4. The general shape of the curves (75% and 100% fault coverage) are similar. The simulation time for 75% of the faults represents only 1.56% of the equivalent testing time at full fault coverage. There is a large saving time, and the procedure is able to select the proper LFSR.
Figure 5.4 - Number of Test Patterns at Different Fault Coverage (c880.isc).

5.4 REDUCED TEST METHOD BASED ON REWIRING

The idea of the Reduced Test Method based Linear Codes has been extended to verification testing based on rewiring, thus generating the Reduced Test Method based on Rewiring. The test patterns generated by the LFSR are distributed to the CUT by the rewiring of the CUT inputs. This same rewiring is be simulated, so that the CUT being simulated receives equivalent test patterns as the real circuit. In the real circuit the rewiring is performed by BILBO structures, or through the use of multiplexers.
Fault simulation of the an \((n,w)\)CUT can be performed using a \(p\)-stage LFSR, with primitive polynomial \(f(x)\) of degree \(p\) \((w \leq p \leq n)\). The value of \(p\) correspond to the minimum number of sets of inputs, which can be obtained using the procedure presented in the previous chapter. The CUT inputs that would be rewired together during testing will receive the same input signal during fault simulation.

As it can be verified, the LFSR with \(p\) stages is usually smaller than the \((n,k)\)LFSR used in the pseudo-exhaustive testing based on Linear Codes.

For each test pattern generated by the LFSR, and distributed to the CUT inputs, the CUT is fault simulated and the detected faults are marked. This process continues until all faults have been marked. This process can be repeated for other \(f(x)\) and/or other seeds \(S_0(x)\), and the \(f(x)\) resulting in the shortest testing time is selected to be implemented in the real circuit. This process is very similar to the Reduced Test Method based on the Linear Codes.

To fault simulate the CUT with its inputs rewired during testing, a new modified version of the \(fsimm.ou\) has been developed. The new version of the \(fsimm.ou\), called \(fsimw.ou\), takes as input files the circuit file description and the dependence matrix file of the CUT. The dependence matrix input file must be similar to the output file generated by the CONVERT. The \(fsimw.ou\) also asks for the same information asked by the \(fsimm.ou\), such as number of undetectable faults, seed \(S_0(x)\), and the number of test patterns to be fault simulated.
The *Reduced Test Method based on Rewiring*, using the *fsimw.ou*, has been applied to many circuits, and Table 5.10 shows the results obtained when the benchmark c880.isc is used.

It can be seen from Table 5.10 that the best result using this method is 15.3% shorter than using the *Reduced Test Method based on Linear Codes* of Table 5.4. Also, the average testing length using this approach is 15.7% shorter than using the *Reduced Test Method based on Linear Codes*.

Table 5.10 also presents the corresponding number of test patterns for 95% fault coverage. For this set of data, the correlation factor is 0.79. It can be seen that even at 95% fault coverage, the number of test patterns is significantly less than full fault simulation.

Table 5.10 - Reduce Test Method Based on Rewiring Applied to c880.isc.

<table>
<thead>
<tr>
<th>Seed</th>
<th>( f(x) )</th>
<th>( 1 + x + x^2 + x^3 + x^4 )</th>
<th>95%</th>
<th>100%</th>
<th>( 1 + x^1 + x^2 + x^4 + x^5 )</th>
<th>95%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 1 )</td>
<td></td>
<td>24000</td>
<td>960</td>
<td>18656</td>
<td>608</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 1 + x )</td>
<td></td>
<td>22784</td>
<td>864</td>
<td>12064</td>
<td>512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 1 + x + x^2 + x^3 + x^4 )</td>
<td></td>
<td>17920</td>
<td>512</td>
<td>16608</td>
<td>576</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 1 + x + x^2 + x^3 + x^4 + x^5 )</td>
<td></td>
<td>22784</td>
<td>416</td>
<td>12704</td>
<td>384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 )</td>
<td></td>
<td>17984</td>
<td>384</td>
<td>5696</td>
<td>288</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 )</td>
<td></td>
<td>17984</td>
<td>384</td>
<td>14624</td>
<td>416</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^{10} )</td>
<td></td>
<td>17984</td>
<td>384</td>
<td>14624</td>
<td>416</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Although this method usually results in shorter number of test patterns than the *Reduced Test Method based on Linear Codes*, it should be noted that it requires rewiring of the CUT inputs during test. For each circuit, an analysis of the rewiring problems (such as area) and testing time should be considered in the choice between both reduced testing methods.

### 5.5 Probabilistic Analysis

The *fault detection probability* \( p \) was defined in equation (3.20) of chapter 3 as

\[
p = \frac{d_f}{u}
\]

where the detectability \( d_f \) of a fault is the number of test patterns, from the set \( u \) that detects the fault.

Analysis is now performed to obtain the probability of detection of a fault \( f \) (of detection probability \( p \)) within \( m \) test vectors. Initially, the analysis is performed considering that the test vectors being generated are purely random. The probability that the first vector detects the fault, \( p_r(1) \), is simply \( p \). The probability that the fault is detected in the second test pattern, \( p_r(2) \), is the probability that the fault was not detected in the first test, multiplied by the probability of being detected in the second, or
Since the detection probabilities in equation (5.3) are disjoint, the probability of detecting fault $f$ within $m$ test patterns is

$$p_f = \sum_{i=1}^{m} p_f(i) \quad (5.4)$$

which, after some manipulation is

$$p_f = 1 - (1-p)^m \quad (5.5)$$

Expression (5.5) is found to be equivalent to 1 minus the probability of the fault not being detected until vector $m$. This expression is the probability that a fault $f$ with detection probability $p$ is detected within $m$ test vectors.
From expression (5.5), \( m \) is found to be

\[
m = \frac{\log(1-p_f)}{\log(1-p)}
\]  

which means that \( m \) is the minimum number of test patterns to be applied to assure that a fault \( f \) of detection probability \( p \) is detected with detection probability \( p_f \).

Expression (5.5) have been obtained under the assumption that the LFSR, which supplies the test patterns, acts as a random pattern generator. Actually the LFSR performs the pseudo-random generation, equivalent to a sampling without replacement process, where the random generation is equivalent to a sampling with replacement. Since each test pattern appears only once in the period, expression (5.5) must be reevaluated.

The probability that the first vector detects the fault is \( p \). The probability that the second vector detects the fault is the probability that the fault was not detected by the first vector multiplied by the new fault detection probability. The new fault detection probability is \( d_f \) over the \( u-1 \) remaining possible vectors. The detection probabilities are
Since these probabilities are disjoint, the probability of detecting fault $f$ within $m$ test patterns is

$$P_f = \sum_{i=1}^{m} p_f(i)$$  \hspace{1cm} (5.8)

or

$$P_f = \sum_{i=1}^{m} \left( 1 - \sum_{k=1}^{i-1} p_f(k) \right) * \frac{d_f}{u-i+1}$$  \hspace{1cm} (5.9)

It can be seen that this expression is more complex than expression (5.5), and $m$, equivalent to expression (5.6), must be obtained numerically.

Expressions (5.5) and (5.9) have been simulated for the same conditions and their probabilities of detection are very close. As an example, consider Figure 5.5, where $p = 0.102$. It can be seen that the two probabilities are very close, where their maximum difference is about 1%. The small difference between the random and pseudo-random modeling, indicates that for simplicity, the random modeling can be
used to calculate probabilities of detection when LFSR is used. This simplistic approach should be used only as a rough estimate.

Figure 5.5 - Comparison of Probability of Detection Between Random and Pseudo-Random Processes.

Expressions (5.5) and (5.9) consider only one fault being fault simulated, and the analysis should be extended to consider more than one fault being fault simulated at the same time. Consider initially that the CUT has δ faults, each of them with fault detection probability \( p \). Assume also that the faults are independent: the occurrence of one of them does not depend on the occurrence of another fault.
The probability that \( \delta \) faults are detected within \( m \) test patterns based on the random test patterns is

\[
P_d = (p_r)^\delta
\]  \hspace{1cm} (5.10)

If (5.5) is substituted in (5.10), then, \( p_d \) becomes the probability of detecting all \( \delta \) faults, each of fault detection probability \( p \), within \( m \) test patterns, based on pseudo-random test patterns.

Figure 5.6 presents the effects of the number of faults in the probability of detection \( p_d \), where the fault detection probability is 25/256.

---

Figure 5.6 - Comparison of Probability of Detection \( p_d \) for Different Number of Faults.
The graph of Figure 5.7 shows the effects of fault detection probability $p$ in the probability of detection $p_d$, where in both cases the same number of faults is considered.

![Graph showing probability of detection $p_d$ for different detection probabilities $p$.](image)

**Figure 5.7 - Comparison of Probability of Detection $p_d$ for Different Detection Probabilities $p$.**

Given that the fault detection probability $p$ is fixed and independent of the testing type (exhaustive or pseudo-exhaustive) it must now be justified that the Reduced Test Method is more efficient when applied to pseudo-exhaustive testing, than on exhaustive testing.

Consider Figure 5.8, which illustrates the test set representation to be considered. The set $k$ corresponds to all test patterns generated by an $(n,k)$LFSR, and
set $u$ corresponds to the test patterns generated by an LFSR implementing a primitive polynomial $f(x)$. It is clear that the set $k$ is a sub-set of the set $u$.

![Figure 5.8 - Test Set Representation.](image)

Since the sub set $k$ contains all test patterns necessary to detect all faults, it does not seem to be appropriate to perform the search (fault simulation) on the set $u$. Also, in the worst case the fault simulation can exhaust subset $k$ faster than set $u$.

Consider, for this analysis, the pseudo-random model of the LFSR, and also, consider the case of low fault detection probability. Figure 5.9 shows the typical probability detection for low fault detection probability and pseudo-random model of a LFSR. In Figure 5.9 fault detection probability $p$ is $5/256$ (or $20/1024$), set $k$ is $2^8$ (254) and set $u$ is $2^{10}$ (1024). It can be verified that the smaller detection probability $p$ the larger the difference between the probability of detections $p_j$ of both $u$ and $k$. 
It can be observed that the probability of detection $p_f$ reaches the top value 1 much faster for the sub-set $k$ (pseudo-exhaustive testing) than for the set $u$ (exhaustive testing). Before the value 1 is reached, for the same number of test patterns applied, the probability of detection $p_f$ for sub-set $k$ is higher than for the set $u$. For instance, in Figure 5.9, the probability of detection $p_f=0.8$ for sub-set $k$ and the probability of detection $p_f=0.7$ for set $u$ after applying 'y' test patterns.

Based on these observations, the *Speed-up* factor can be defined as the ratio of probability of detection $p_f$ from set $u$ over the probability of detection $p_f$ from sub-set $k$, or
The Speed-up factor corresponding to the graph of Figure 5.9 is presented in the form of a graph in Figure 5.10. The Speed-up factor is dependent on the number of test patterns being applied, as is illustrated by the example of Figure 5.10.

\[ S_p = \frac{P_f(u)}{P_f(k)} \]  

\[(5.11)\]

The Speed-up factor starts at its minimum of 1, where the probability of detection of a fault is the same if sub-set \(k\) or set \(u\) is used. The Speed-up goes to its maximum, towards the exhaustion of sub-set \(k\), and decreases again to 1 towards the
exhaustion of set $u$. Observe that no more faults are detected from sub-set $k$ after its exhaustion (all faults have been detected), and there are still faults to be detected from set $u$.

It can be observed that the faster the Speed-up goes to its maximum, the faster all faults (based on sub-set $k$) are detected, also, the smaller $p$, the larger Speed-up.

The Speed-up is used to illustrate the advantage (and effectiveness) of using pseudo-exhaustive testing, compared to exhaustive testing to detect all faults using fault simulation.

Another important point to mention related to the reduced test methods is the aliasing. For most practical circuits, the value of the fault detection probability $p$ is usually low, mostly below 0.5. Based on the results of Chapter Three (Graph of Figure 3.14), for $p<0.5$, reduction in the number of test patterns can result in reduced aliasing. Reduction of the aliasing probability is another advantage of the reduced testing methods.

5.6 RELIABILITY - AN INTRODUCTORY ANALYSIS

Reliability represents an important issue for BIST circuits, and this section presents an introduction to changes in reliability due to the addition of BIST circuitry, such as the Reduced Test Methods.

In this introductory analysis, for simplicity, consider the failure rate $\lambda$ being constant and independent of time (exponential distribution). Consider that the
components analyzed are on the normal operation life, since no debugging exists (the circuit works, or not) and neither wear-out (by the same reason).

It is assumed that 90% of the circuits should be working (no failure) after 5 years. For exponential distribution, the failure density function $f(t)$ and the failure rate $\lambda$ are related by [Billinton 83]

$$f(t) = \lambda e^{-\lambda t}$$

(5.12)

where $t$ is time (here considered in years). Then, after 5 years, and considering 90% of the circuits without failures

$$0.9 \lambda = \lambda e^{-\lambda \times 5}$$

obtaining $\lambda = 0.021$ [failures/year].

The Mean Time To Failure (MTTF) is given as $1/\lambda$, or

$\text{MTTF} = 47.46$ [years]. MTTF is the average time a system takes to fail.

Also, consider that the failure rate $\lambda$ of the circuit is related (dependent) on the number of elements (or gates) of the circuit. Under this model, if a circuit has an overhead due to BIST, the failure rate $\lambda$ increases proportionally, to a new failure rate $\lambda_{\text{BIST}}$.

Taking as an application example, the ISCAS85 c7552.isc (ALU and Selector) has 207 inputs and is composed of 3827 gates (elements). Turning the input and output latches into BILBO (see Figures 1.13 and 1.14), to make it BIST, corresponds
to an additional of 3 gates per input/output, plus 4 gates due to multiplex at the
beginning of each chain, and plus 5 gates of the Exclusive-OR of the LFSR feedback.

Then the overhead is

\[
\begin{align*}
\text{input:} & \quad 3 \times 207 + 5 + 4 = 630 \\
\text{output:} & \quad 3 \times 108 + 5 + 4 = 333 \\
& \quad 963
\end{align*}
\]

which corresponds to an overhead of 25% on the c7552.isc.

The new failure rate \( \lambda_{\text{BIST}} \) is then 25% higher than its original failure rate \( \lambda \),
or

\[
\lambda_{\text{BIST}} = 1.25 \lambda = 0.02625 \text{ [failure/year]},
\]

from which

\[
\text{MTTF}_{\text{BIST}} = 38.1 \text{ [years]}, \text{ corresponding to a decrease of } 19.7\% \text{ of the original MTTF.}
\]

Under the new failure rate \( \lambda_{\text{BIST}} \), after 5 years 87.7% of the components are
expected to be fault free. This represents a decrease of 25% in the number of
components working after 5 years, when compared to the original circuit.

Figure 5.11 shows the failure density function for both \( \lambda \) and \( \lambda_{\text{BIST}} \). Although
the failure rates are different (\( \lambda_{\text{BIST}} > \lambda \)), their \( f(t) \) will assume the same value at 42.4
years, and after that time \( f(t) > f(t)_{\text{BIST}} \).

The area under the failure density function for up to a certain time corresponds to the \textit{Cumulative Fault Probability} \( Q \). The Cumulative Fault Probabilities for 42.4 years are .592 and .675 for \( \lambda \) and \( \lambda_{\text{BIST}} \), respectively, which corresponds to an
increase of 13.9%. Table 5.11 presents the Cumulative Failure Density of the
c7552.isc for \( \lambda \) and \( \lambda_{\text{BIST}} \) for selected values, including 44.2 years.
It can be seen that initially the difference between the Cumulative Fault Probability of $\lambda$ and $\lambda_{BIST}$ is large, and as time increases, the difference decreases.

The actual changes in failure rate $\lambda$ may not be such as presented, which is just an introduction. The proper behavior of the failure rate when BIST is implemented require a deeper analysis, and it is suggested as a future work.

![Failure Density Function for c7552.isc.](image-url)
Table 5.11 - Cumulative Fault Probability for c7552.isc.

<table>
<thead>
<tr>
<th>time (years)</th>
<th>Q</th>
<th>Q_{HHT}</th>
<th>difference (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.024937</td>
<td>0.031091</td>
<td>24.67411</td>
</tr>
<tr>
<td>2</td>
<td>0.045244</td>
<td>0.056262</td>
<td>24.35164</td>
</tr>
<tr>
<td>3</td>
<td>0.0655128</td>
<td>0.080780</td>
<td>24.03257</td>
</tr>
<tr>
<td>4</td>
<td>0.084600</td>
<td>0.104664</td>
<td>23.71687</td>
</tr>
<tr>
<td>5</td>
<td>0.103666</td>
<td>0.127929</td>
<td>23.40452</td>
</tr>
<tr>
<td>6</td>
<td>0.122337</td>
<td>0.150591</td>
<td>23.09550</td>
</tr>
<tr>
<td>7</td>
<td>0.140619</td>
<td>0.172666</td>
<td>22.78979</td>
</tr>
<tr>
<td>8</td>
<td>0.158522</td>
<td>0.194169</td>
<td>22.48736</td>
</tr>
<tr>
<td>9</td>
<td>0.176052</td>
<td>0.215115</td>
<td>22.18820</td>
</tr>
<tr>
<td>10</td>
<td>0.193218</td>
<td>0.235518</td>
<td>21.89227</td>
</tr>
<tr>
<td>20</td>
<td>0.346433</td>
<td>0.412623</td>
<td>19.10610</td>
</tr>
<tr>
<td>30</td>
<td>0.470627</td>
<td>0.548840</td>
<td>16.61876</td>
</tr>
<tr>
<td>40</td>
<td>0.571297</td>
<td>0.653607</td>
<td>14.40764</td>
</tr>
<tr>
<td>44.2</td>
<td>0.592471</td>
<td>0.674917</td>
<td>13.91559</td>
</tr>
<tr>
<td>50</td>
<td>0.652898</td>
<td>0.734187</td>
<td>12.45045</td>
</tr>
</tbody>
</table>
6. CONCLUSIONS AND FUTURE WORK

Testing of integrated circuits is becoming very expensive, and accounts for nearly a third of total cost [Levitt 92]. The high density and complexity of integrated circuits make testing not only expensive, but also complex and time-consuming.

BIST makes the testing less complex, because the external equipment can be simplified, and the testers are simpler and less expensive. Although the testing can be simplified, the testing time still remains long. Pseudo-exhaustive testing methods are often used to design BIST circuits. Pseudo-exhaustive testing reduces the testing time, although the time still needs further reduction.

This work was dedicated to investigate reductions in testing time, based on pseudo-exhaustive techniques. Initially the pseudo-exhaustive testing techniques were extended and improved. As presented in Chapter Four, the verification testing has been complemented with the Automatic Generation of Dependence Matrix, and its respective program CONVERT. This program, implemented in C language, tracks the input dependencies of each circuit output, and provides as result the Dependence Matrix.

Also presented in Chapter Four is a method to obtain the proper sets of circuit inputs, to be wired together during test. This procedure has been implemented also in C language (LAST). The sets of inputs are obtained based on the circuit topology.
Both CONVERT and LAST provide excellent results on the circuits. Their listing and User's Guide are supplied in Appendices A and B, respectively.

Based on pseudo-exhaustive testing techniques, the Reduced Test Methods have been developed. The basic idea of the Reduced Test Methods is to design a proper LFSR leading to a small number of test patterns applied consecutively to the circuit, such that all faults are detected.

In this method the test patterns are applied to the Circuit Under Test only until all faults have been detected. This corresponds to a small fraction of the pseudo-exhaustive testing time, since only a small sub-set of the pseudo-exhaustive test patterns is applied. This minimum number of consecutive test patterns, that detect all faults, is obtained from simultaneous simulation of the LFSR (test patterns generator) and fault simulation of the Circuit Under Test. In the fault simulation process, for each test pattern generated by the LFSR, the Circuit Under Test is fault simulated, and the detected faults are marked. The process is stopped when all faults have been marked (detected). The number of test patterns obtained during fault simulation, corresponds to the minimum number of consecutive test patterns that detect all faults in the actual circuit.

This procedure is repeated for different LFSR (implementing different characteristic polynomial), and the LFSR resulting in the short testing time is selected for real circuit implementation. The testing time can be greatly reduced, since each LFSR would generate a different test pattern sequence, each requiring different testing time to detect all faults.
As a refinement, instead of simulating the LFSR candidates for full fault coverage, they are fault-simulated for only a fraction of the total number of faults. Then, a few of those LFSR resulting in short testing time are fault-simulated for full fault coverage. This process works as a two-section selection, with only those performing better in the first section (fraction of the faults) take part in the second section (full fault simulation). This idea reduces the overall testing time, since only a small part of the LFSR candidates is fully fault-simulated. This concept was proved to be successful by the correlation study presented.

The Reduced Test Methods, including the refinements, have been applied to many circuits, including some ISCAS85 benchmark circuits, with excellent results. For large circuits, the testing time is reduced to less than 1% of the pseudo-exhaustive testing time.

A small disadvantage of the Reduced Test Methods is the simulation time required to the proper selection of the LFSR to be implemented. This disadvantage is practically eliminated by implementation of a two-stage fault simulation. Also, it should be noted that the fault simulation process can be carried along the design of each section of the integrated circuit.

Although the overhead is the same as pseudo-exhaustive testing, the testing time is considerably lower, and provides the same fault coverage. No extra area overhead is required since no circuitry is added, there are just proper selection of the LFSR to be implemented and the adequate limit on the number of test patterns.
Since no extra circuitry is added, when compared to pseudo-exhaustive testing, there is no alteration in the circuit reliability. Change in reliability would occur if the CUT with no additional testing circuitry is compared to the CUT implementing the Reduced Test Methods (or just pseudo-exhaustive testing). The reliability degradation of the circuit with BIST compared with no BIST seems reasonable low.

Detailed investigation of reliability issue is suggested for a future work. Since this work considered the stuck-at fault model, it is suggested as a future work, an investigation of the Reduced Test Method applied to the stuck-on, stuck-open and other fault models. It is also suggested as a future work, an investigation of the Reduced Test Method applied to sequential circuits.
REFERENCES


A1. CONVERT USER'S GUIDE

CONVERT is a program developed in C programing language to convert files from ISCAS85 benchmark format to TESS format. The program also obtain the Dependence Matrix.

The input file, name.isc, must follow the description adopted by the ISCAS85 benchmark circuits (see section A3). Here name is the name of the file to be processed.

CONVERT generates the equivalent circuit in the TESS format, name.tes (see section A3). CONVERT also provides the Dependence Matrix, name.w (see section A3).

The program can be run as follows:

convert name.isc

and the output files will be generated automatically.

The user must be aware that this program does not provide any protection against improper circuit description, and the results are not guaranteed for such cases. The program listing is given in section 2.
A2. CONVERT LISTING

/*

This program is used to convert the files in the ISCAS85 format to the TESS format.
The program takes as input the circuit description file (name.isc) in ISCAS85 format, and generates the corresponding output file (name.tes) in TESS format. The program also generates the partition table of the circuit (name.w), providing the total number of inputs and the output dependency of each output.

Tales Cleber Pimenta
Ohio University
ECE - Stocker Center
Athens - OH - 45701
tel (614) 594 5410
fax (614) 593 0007
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define LONG 8000
#define WIDE 8
#define BYTE 30
FILE *input, *output;
int flag, count;
char file[20];
struct fanlist
{
    int in;
    struct fanlist *next;
};
struct wire
{
    int new;
    int addr;
    char name[10];
    char type[5];
    char from[10];
    short fanin;
    struct fanlist *flist;
    struct wire *next;
};
struct wire *line, *oldline, *head, *topinpt;
main(argc, argv)
int argc;
char *argv[];
char filein[20], fileout[20], ch, *s1, *s2;
/* Ask the name of the input file - if not provided in the
command line */
if (argc == 1)
{
    print("\nWhat is the name of the input file? ");
    scanf("%s",filein);
}
else
    strcpy (filein,***argv);
input=fopen(filein,"r");
if (input == NULL)
{
    print("\n
 *** Can't open the %s file ***
",filein);
    exit (0);
}
/* Obtain the output file */
s2=fileout;
for (s1 = filein;*s1 != '.' && *s1 != '\0'; s1++)
    *s2++=*s1;
*s2='\0';
strcpy(file,fileout);
strcat(fileout,".tes");
output = fopen(fileout,"w");
if (output == 'NULL')
{
    print("\n
 *** Can't open the output file ***
\n\n");
    output = stdout;
}
flag = 0;
count = 0;
/* Disregard the useless information lines */
while ((ch=getc(input)) != EOF)
{
    ungetc (ch, input);
    if (ch==')'
        fscanf(input,"%*[^n]%c");
    else
    {
        line = (struct wire *)malloc(sizeof(struct wire));
        count++;
        if (line == NULL)
        {
            print("\n <<< Out of memory >>>");
            exit (0);
        }
    }
    /* Call the function to read each line of information */
    getline();
    /* Placing the 'inpt' lines first and then the remaining
if (flag == 0)
{
head = line;
oldline = line;
topinpt = line;
flag = 1;
}
else
{
if (strcmp(line->type,"inpt") == 0)
{
if (oldline->next == topinpt->next)
{
oldline->next = line;
topinpt->next = line;
oldline = line;
topinpt = line;
}
else
{
line->next = topinpt->next;
topinpt->next = line;
topinpt = line;
}
}
else
{
oldline->next = line;
oldline = line;
}
}
/* Provide new address for the sorted lines */
setnew();
/* Correct the 'from' lines -> put as 'gate' */
setfrom();
/* Adjust the 'fan in' lines according to the new addresses */
setfan();
/* Print the headings and the circuit lines */
putline();
fclose(input);
fclose(output);
/* Find the input dependency of each output */
findw();
/* Function to read each information line */
ggetline()
{
short aux, index;
struct fanlist *list, *oldlist;
/* Read the address, name and type of each line */
fsnaf(input,"%d %s %s", line->addr, line->name, line->type);
/* Collect the 'inpt' lines */
if (strcmp(line->type, "inpt") == 0)
{
    fscanf(input,"%*[\n]*c");
    line->fanin = 0;
    line->flist = NULL;
}
else
{
    /* Collect the 'from' lines */
    if (strcmp(line->type, "from") == 0)
    {
        fscanf(input,"%s%*[\n]*c", line->from);
        line->fanin = 0;
        line->flist = NULL;
    }
    else
    {
        /* Collect the remaining types of lines */
        fscanf(input,"%hd %hd%*[\n]*c", &aux, &line->fanin);
        /* Obtain the fan in list of each gate line */
        for (index = 0; index < line->fanin; index++)
        {
            list = (struct fanlist *)malloc(sizeof(struct fanlist));
            if (list==NULL)
            {
                print("\n\n << Out of memory >>\n");
                exit(0);
            }
            else
            {
                fscanf(input,"%d", &list->in);
                if (index==0)
                {
                    list->next = NULL;
                    line->flist = list;
                    oldlist = list;
                }
                else
                {
                    list->next = NULL;
                    oldlist->next = list;
                    oldlist = list;
                }
            }
        }
        fscanf(input,"%*[\n]*c");
    }
}
setnew()
{
  int aux;
  line = head;
  for (aux = 1; aux <= count; aux++)
  {
    line->new = aux;
    if (aux == count)
      break;
    line = line->next;
  }
}

setfrom()
{
  struct wire *ptr;
  struct fanlist *list;
  int aux, index;
  line = head;
  for (aux = 1; aux <= count; aux++)
  {
    if (strcmp(line->type, "from") == 0)
    {
      ptr = head;
      for (index = 1; index <= count; index++)
      {
        if (strcmp(ptr->name, line->from) == 0)
        {
          /* Open one fan in position for the new mode of 'from'
           */
          list = (struct fanlist *)malloc(sizeof(struct fanlist));
          if (list == NULL)
          {
            print("\n\n <<< Out of Memory >>> ");
            exit(0);
          }
          else
          {
            list->next = NULL;
            line->flist = list;
            list->in = ptr->addr;
            line->fanin = 1;
            break;
          }
        }
      }
      if (index == count)
        break;
    }
  }
ptr=ptr->next;
}
}
if (aux == count)
  break;
line = line->next;
}

/* Adjust the fan in number according to the new address */
setfan()
{
struct wire *ptr;
struct fanlist *list;
int i, index, aux;
line = head;
for (aux=1;aux<=count;aux++)
{
  if (line->fanin>0)
  {
    list = line->flist;
    for (index=1;index<=line->fanin;index++)
    {
      ptr = head;
      for (i=1;i<=count;i++)
      {
        if (list->in == ptr->addr)
        {
          list->in = ptr->new;
          break;
        }
        if (i == count)
          break;
      ptr = ptr->next;
    }
    if (index == line->fanin)
      break;
    list = list->next;
  }
  if (aux == count)
    break;
  line = line->next;
}

/* Print the headings and the new circuit description */
putline()
{
int i, index, aux;
char c[5];
struct fanlist *list;
line = head;
/* Print the heading */
fprintf(output,"faultlist\n");
for (aux=0;aux<=1;aux++)
{
  i=0;
  for (index=1;index<count;index++)
  {
    fprintf(output,"(%d %d ) ",index,aux);
    i++;
    if (((i==9) || (index==count-1))
    {
      fprintf(output,"\n");
      i=0;
    }
  }
}
fprintf(output,"circuit");
/* Print the new circuit description */
for (aux=1;aux<count;aux++)
{
  /* Print the 'inpt' lines as INPUT */
  fprintf(output,"\n%d ",line->new);
  if (strcmp(line->type,"inpt")==0)
    fprintf(output,"INPUT");
  else
    {
      /* Convert the 'from' and 'buff' lines to FAN */
      if (strcmp(line->type,"from")==0 ||
          strcmp(line->type,"buff")==0)
        fprintf(output,"FAN");
      else
        {
          /* Put the remaining lines names in capital letters */
          strcpy(c,line->type);
          for (i=0;c[i] != '\0';i++)
            fprintf(output,"%c",toupper(c[i]));
        }
    }
  /* Print the fan in listing of each gate */
  if (line->fanin > 0)
  {
    list = line->flist;
    for (index=1;index<=line->fanin;index++)
    {
      fprintf(output," %d",list->in);
      if (index == line->fanin)
        break;
      list = list->next;
    }
  }
  fprintf(output," <1 1> ;");
if (aux == count)
    break;
    line=line->next;
}

/* Find the input dependency of each output */
findw()
{
    "fileout[20], ch, type[7];
    int i, j, index, out[LONG], inpt, outpt, addr, fan;
    long line[LONG][WIDE], one, aux1, aux2;
    for (i=0;i<LONG;i++)
    {
        out[i]=0;
        for (index=0;index<WIDE;index++)
            line[i][index]=0;
    }

    /* Read from the previously generated files and generate new file */
    strcpy(fileout, file);
    strcat(fileout, ".w");
    strcat(file, ".tes");
    input = fopen(file, "r");
    if (input==NULL)
    {
        print("\n\n *** Can’t open the %s file ***\n", file);
        exit(0);
    }
    output = fopen(fileout, "w");
    if (output==NULL)
    {
        print("\n\n *** Can’t open the %s file ***\n", fileout);
        output = stdout;
    }
    one=1;
    i=0;
    while ((ch=getc(input)) != EOF)
    {
        if (ch >='0' && ch <= '9')
        {
            ungetc(ch, input);
            fscanf(input, "%d %s", &addr, type);
            if (addr>=LONG)
            {
                print("\n\n *** Too many input lines ***\n");
                exit(0);
            }
        }
    }

    /* Analysis of INPUT lines */
    if (strcmp(type, "INPUT") == 0)
i++;
aux1=((addr-1) % BYTE);
aux2=((addr-1) / BYTE);
line[addr][aux2] = (one << (aux1));
    fscanf(input,"%*[\n]%*c");
}
else
{
    /* Analysis of the single input lines */
    if (strcmp(type,"FAN")==0 || strcmp(type,"NOT")==0)
    {
        fscanf(input,"%d%*[\n]%*c", &fan);
        for (index=0;index<WIDE;index++)
            line[addr][index]=line[fan][index];
        out[addr]=1;
        out[fan]=0;
    }
    else
    {
        /* Analysis of the remaining type of lines */
        while (((ch=getc(input)) != '<')
            {
                if (ch>='0' && ch<='9')
                    {
                        ungetc(ch,input);
                        fscanf(input,"%d",&inpt);
                        for (index=0;index<WIDE;index++)
                            line[addr][index] = line[addr][index] |
                        line[inpt][index];
                        out[addr]=1;
                        out[inpt]=0;
                    }
                fscanf(input,"%*[\n]%*c");
            }
    }
    else
        fscanf(input,"%*[\n]%*c");
}
fclose(input);
input = fopen(file,"a");

/* Determination of the maximum output weight */
print("\n\nNumber of inputs %d",i);
print("\n\nOUTPUT\t(W)\tDEPENDENCIES\n");
fprintf(output,"\n\nNumber of inputs %d",i);
fprintf(output,"\n\nOUTPUT\t(W)\tDEPENDENCIES\n");
count=addr;
for (i=1;i<=addr;i++)
    {
if (out[i] == 1)
{
    count++;
    fprintf(input,"\n%d OUTPUT %d <0 0> ;", count, i);
    output = 0;
    print("\n%d", count);
    fprintf(output,"\n%d", count);
    for (index = 0; index < WIDE; index++)
    {
        for (j = 1; j <= BYTE; j++)
        {
            aux1 = (one << (j-1));
            aux2 = (aux1 & line[i][index]);
            if (aux2 != 0)
            {
                outpt++;
            }
        }
        print("\t(%d) : 	", outpt);
        fprintf(output, "\t(%d) : 	", outpt);
    }
}
/*
Determination of the input dependencies of each output
*/
flag = 0;
for (index = 0; index < WIDE; index++)
{
    for (j = 1; j <= BYTE; j++)
    {
        aux1 = (one << (j-1));
        aux2 = (aux1 & line[i][index]);
        if (aux2 != 0)
        {
            print(" %d", (j + BYTE*index - 1));
            fprintf(output, " %d", (j + BYTE*index - 1));
            flag++;
            if (flag == 15)
            {
                print("\n\n\t\t");
                flag = 0;
            }
        }
    }
}
print("\n\n");
fclose(input);
fclose(output);
A3. CONVERT INPUT AND OUTPUT FILES

Input File:

Each line of the ISCAS85 circuit description consists of

<Number> <Name> <Type> <Outputs> <Inputs> <Listing> <Faults>.

Where

<Number>  gate output or fanout number,
{Name>  gate output or fanout name,
<Type>  type of gate, or indicate if it is fanout node,
<Outputs>  number of gate fanout,
<Inputs>  number of gate fanins,
<Listing>  listing of inputs, by its Number or Name,
<Faults>  list of stuck-at faults to be considered.

In the ISCAS85 circuit description, the comment lines start with an asterisk, and are placed at the beginning of the circuit description.

As an example, the ISCAS85 c17.isc circuit file description is given, and its circuit is presented in Figure A1.
c17.isc

*c17 iscas example (to test conversion program only)
*---------------------------------------------------
*
* total number of lines in the netlist ............... 17
* simplistically reduced equivalent fault set size = 22
* lines from primary input gates ....... 5
* lines from primary output gates ....... 2
* lines from interior gate outputs ...... 4
* lines from ** 3 ** fanout stems ... 6
* avg_fanin = 2.00, max_fanin = 2
* avg_fanout = 2.00, max_fanout = 2
*
*
1  1gat inpt  1  0  >sal
2  2gat inpt  1  0  >sal
3  3gat inpt  2  0  >sa0  >sal
8  8fan from   3gat  >sal
9  9fan from   3gat  >sal
6  6gat inpt  1  0  >sal
7  7gat inpt  1  0  >sal
10 10gat nand 1  2  >sal
    8
11 11gat nand 2  2  >sa0  >sal
9  6
14 14fan from   11gat  >sal
15 15fan from   11gat  >sal
16 16gat nand 2  2  >sa0  >sal
    2
20 20fan from   16gat  >sal
21 21fan from   16gat  >sal
19 19gat nand 1  2  >sal
15  7
22 22gat nand 0  2  >sa0  >sal
10  20
23 23gat nand 0  2  >sa0  >sal
21  19
Output File:

Each line of the TESS circuit description consists of

\(<Number> \ <Type> \ <Listing> \ <s0 \ s1>\)

Where

\(<Number>\)  gate output or fanout number,
\(<Type>\)     type of gate, or indicate if it is fanout node,
\(<Listing>\)  listing of inputs, by its Number,
\(<s0 \ s1>\)   s0 and s1 means stuck-at-0 and stuck-at-1, respectively, and a 1 is placed if the s0 or s1 fault should be considered, otherwise a 0 is placed.

The TESS circuit description requires that the fault list be placed before the circuit description, with one list for the stuck-at-0 faults and another for the stuck-at-1 faults. There is no comment lines on this format. The c17.tes circuit is presented in Figure A1, and the equivalent TESS circuit description is given below.
c17.ics

faultlist
(1 0) (2 0) (3 0) (4 0) (5 0) (6 0) (7 0) (8 0) (9 0)
(10 0) (11 0) (12 0) (13 0) (14 0) (15 0) (16 0) (17 0)
(1 1) (2 1) (3 1) (4 1) (5 1) (6 1) (7 1) (8 1) (9 1)
(10 1) (11 1) (12 1) (13 1) (14 1) (15 1) (16 1) (17 1)
circuit
1 INPUT <1 1> ;
2 INPUT <1 1> ;
3 INPUT <1 1> ;
4 INPUT <1 1> ;
5 INPUT <1 1> ;
6 FAN 3 <1 1> ;
7 FAN 3 <1 1> ;
8 NAND 1 6 <1 1> ;
9 NAND 7 4 <1 1> ;
10 FAN 9 <1 1> ;
11 FAN 9 <1 1> ;
12 NAND 2 10 <1 1> ;
13 FAN 12 <1 1> ;
14 FAN 12 <1 1> ;
15 NAND 11 5 <1 1> ;
16 NAND 8 13 <1 1> ;
17 NAND 14 15 <1 1> ;
18 OUTPUT 16 <0 0> ;
19 OUTPUT 17 <0 0> ;
Figure A1 - c17.isc (ISCAS85).

Figure A2 - c17.tes (TESS).
Dependence Matrix

The CONVERT program provides also as output the Dependence Matrix of the input circuit file. The Dependence Matrix uses the same gate number as the TESS format (output file).

The format of each dependence is

\(<Output> <W> <Dependencies>\)

where

\(<Output>\) is the output Number,

\(<W>\) is the output weight (number of inputs that the output depends on),

\(<Dependencies>\) list of inputs that the output depends on.

As an example the Dependence Matrix of the c17.isc, is presented below:

\[\begin{array}{c}
\text{c17.w} \\
\text{Number of inputs 5} \\
\text{OUTPUT} & (W) & DEPENDENCIES \\
18 & (4) : & 0 1 2 3 \\
19 & (4) : & 1 2 3 4 \\
\end{array}\]
B1. LAST USER'S GUIDE

LAST is a program developed in C programing language to obtain the Partitioned Dependence Matrix from the Dependence Matrix of a given circuit.

The input file, name.w, must have the same format as the output file of CONVERT (see section A3 in Appendix A).

LAST generates as an output the Partitioned Dependence Matrix, name.out (see section B3).

The program can be run as follows:

last name.w

and the output file will be generated automatically.

The user should be aware that LAST does not have any protection against improper Dependence Matrix description (input file), and the results are not guaranteed for such cases. The program listing is provided in section B2.
This is a program for partitioning verification testing matrices.
Testing of outputs that do not depend on the same inputs can be performed at the same time, thus reducing the global testing time. This program takes the dependence matrix and generates the partitioned testing matrix.

The dependence matrix is read from the input file. In the input file, the information before de colon (:) in each line is ignored. Each row of the input matrix (corresponding to each circuit output) is imputed after the colon (:). The next row should be placed after the next colon (:).

Ex:
out1 : 2 5 13
out2 : 1 6 11
out3 : 0 5 6 9 10

The partitioned matrix is presented on the screen. One input is inside parenthesis and the others that should be placed together follows that input. If no other input follows, means that the input is left alone.

Tales Cleber Pimenta
Ohio University
ECE - Stocker Center
Athens OH 45701

*/

#define COLUMN 250
#define ROW 100
#include <stdio.h>
FILE *input, *output;
int r, c, temp, ii, jj, count;
short Edge[COLUMN][COLUMN], Neighbor[COLUMN][COLUMN],
Delet[COLUMN][COLUMN], Tag[COLUMN]; float W[COLUMN][COLUMN];
main(argc, argv)
int argc;
char *argv[];
{
register i, j, k;
char *s1, *s2, ch, filein[20], fileout[20];
short D[ROW][COLUMN];
int w1, w2;
float w;
/* Ask the name of the input file - if not provided (command line) */
if (argc==1)
{
    printf("\nWhat is the name of the input file?  ");
    scanf("%s",filein);
}
else
    strcpy(filein,*++argv);
input=fopen(filein,"r");
if (input==NULL)
{
    printf("\n *** Can't open the %s file *** ",
    filein);
    exit(0);
}
/* Obtain the output file */
s2=fileout;
for (s1=filein,*s1 != '.' && *s1 != '\0';s1++)
    *s2++=*s1;
*s2='\0';
strcat(fileout,".out");
output=fopen(fileout,"w");
if (output==NULL)
{
    printf("\n *** Can't open the %s file *** ",fileout);
    output=stdout;
}
/* Reset the working variables */
for (i=0;i<COLUMN;i++)
{
    Tag[i]=i;
    for (j=0;j<COLUMN;j++)
    {
        Edge[i][j]=0;
        W[i][j]=0.0;
    }
    for (j=0;j<ROW;j++)
        D[j][i]=0;
} /* Read the input file into the matrix D */
temp=0;
r=-1;
while((ch=getc(input))!=EOF)
{
    if (ch=='\n')
        temp=0;
    else
    {
        if (ch=='\:')
        {
            temp=1;
            r++;
        }
else
    {
        if (temp==1)
        {
            ungetc(ch,input);
            fscanf(input, "%d", &c);
            D[r][c]=1;
        }
    }
}

/* Find the largest c (length of the matrix D) */
i=0;
for (c=0; c<COLUMN; c++)
    for (j=0; j<=r; j++)
        if (D[j][c]==1)
            {
                i=c;
                break;
            }
c=i;

/* Initialize Edge (rows that can be placed together) */
count=0;
for (i=0; i<c; i++)
    for (j=i+1; j<=c; j++)
        {
            temp=1;
            for (k=0; k<=r; k++)
                if ((D[k][i]&&D[k][j])==1)
                    {
                        temp=0;
                        break;
                    }
            if (temp==1)
                {
                    Edge[i][j]=1;
                    count++;
                }
        }

/* Find max input */
w=0;
for (i=0; i<=c; i++)
    {
        k=0;
        for (j=0; j<=r; j++)
            if (D[j][i]==1)
                k++;
        if (k>w)
            w=k;
    }

/* Obtain: neighbors and deleted */
neighbor();
/* Obtain the edges weight */
for (i=0;i<c;i++)
  for (j=i+1;j<=c;j++)
    if (Edge[i][j]==1)
      {
      w1=0;
      w2=0;
      for (k=0;k<=r;k++)
      {
        if (D[k][i]==1)
          w1++;
        if (D[k][j]==1)
          w2++;
      }
      W[i][j]=(w1+w2)/w;
    }
/* Choose edge to be deleted, merge edges, correct new neighbors */
while (count>O)
  {
    choose();
    merge();
    if (count>O)
    neighbor();
  }
/* Correct the tags */
for (i=0;i<=c;i++)
  {
    if (Tag[i]!=i)
      {
        j=Tag[i];
        while (Tag[j]!=j)
          j=Tag[j];
        Tag[i]=j;
      }
  }
/* Print the results */
printf("\n\nFor the %s file, the best partition is:\n",filein);
for (i=0;i<=c;i++)
  if (Tag[i]==i)
    {
      printf("\n%d : ",i);
      fprintf(output,"\n%d : ",i);
      for (j=0;j<=c;j++)
        if ((Tag[j]==i)&&(i!=j))
          {
            printf("%d ",j);
            fprintf(output,"%d ",j);
          }
    }
}
printf("
");
fprintf(output,"\n");
}
/* Function to find the common neighbors and edges to be deleted */
neighbor()
{
register i, j, k;
/* Reset of the working variables */
for (i=0;i<=c;i++)
  for (j=0;j<=c;j++)
  {
      Delet[i][j]=0;
      Neighbor[i][j]=0;
  }
/* Find the common neighbors */
for (i=0;i<c;i++)
  for (j=i+1;j<=c;j++)
    if (Edge[i][j]==1)
    {
      for (k=0;k<i;k++)
        if ((Edge[k][i]==Edge[k][j])&&(Edge[k][i]==1))
          Neighbor[i][j]++;
      for (k=i+1;k<j;k++)
        if ((Edge[i][k]==Edge[k][j])&&(Edge[i][k]==1))
          Neighbor[i][j]++;
      for (k=j+1;k<c;k++)
        if ((Edge[i][k]==Edge[j][k])&&(Edge[i][k]==1))
          Neighbor[i][j]++;
/* Find the edges to be deleted */
for (k=0;k<i;k++)
  if (Edge[k][i]==1)
    Delet[i][j]++;
for (k=i+1;k<=c;k++)
  if (Edge[i][k]==1)
    Delet[i][j]++;
for (k=0;k<j;k++)
  if (Edge[k][j]==1)
    Delet[i][j]++;
for (k=j+1;k<=c;k++)
  if (Edge[j][k]==1)
    Delet[i][j]++;
  Delet[i][j]=Delet[i][j]-Neighbor[i][j]-1;
  temp=Delet[i][j];
}
/* Function to choose the edge to be deleted */
choose()
{
register i, j, k;
int maxneigh, maxw, index;
/* First option: minimum number of deleted edges */
for (i=0; i<c; i++)
    for (j=i+1; j<=c; j++)
        if (Edge[i][j]==1)
            {  
                if (Delet[i][j]<temp)
                    {
                        temp=Delet[i][j];
                        index=1;
                        ii=i;
                        jj=j;
                    }
                if (Delet[i][j]==temp)
                    index++;
            }
if (index==1)
    return(0);
/* Second option: maximum number of common neighbors */
maxneigh=0;
for (i=0; i<c; i++)
    for (j=i+1; j<=c; j++)
        if (((Delet[i][j]==temp)&&(Edge[i][j]==1))
            {  
                if (Neighbor[i][j]>maxneigh)
                    {
                        maxneigh=Neighbor[i][j];
                        index=1;
                        ii=i;
                        jj=j;
                    }
                if (Neighbor[i][j]==maxneigh)
                    index++;
            }
if (index==1)
    return(0);
/* Third option: Maximum weight */
maxw=0;
for (i=0; i<c; i++)
    for (j=i+1; j<=c; j++)
        if ((Neighbor[i][j]==maxneigh)&&(Delet[i][j]==temp)&&(Edge[i][j]==1))
            if (W[i][j]>maxw)
                {  
                    maxw=W[i][j];
                    ii=i;
                    jj=j;
                }
return(0);
}    /* Function to merge nodes */
merge()
{  

register i, j, k;
for (i=0; i<ii; i++)
{
    if ((Edge[i][ii]==1) && (Edge[i][jj]==1))
        {  
            Edge[i][ii]=0;
            count--;
        }
    else
        {  
            if ((Edge[i][ii]==0) && (Edge[i][jj]==1))
                {  
                    Edge[i][jj]=0;
                    count--;
                }
            else
                if ((Edge[i][ii]==1) && (Edge[i][jj]==0))
                    {  
                        Edge[ii][i]=0;
                        count--;
                    }
        }
}
for (i=ii+1; i<jj; i++)
{
    if ((Edge[ii][i]==1) && (Edge[i][jj]==1))
        {  
            Edge[ii][i]=0;
            count--;
        }
    else
        {  
            if ((Edge[ii][i]==0) && (Edge[i][jj]==1))
                {  
                    Edge[i][jj]=0;
                    count--;
                }
            else
                if ((Edge[ii][i]==1) && (Edge[i][jj]==0))
                    {  
                        Edge[ii][i]=0;
                        count--;
                    }
        }
}
for (i=jj+1; i<=cii; i++)
{
    if ((Edge[ii][i]==1) && (Edge[jj][i]==1))
        {  
            Edge[ii][i]=0;
            count--;
        }
}
\[ W[jj][i] = W[jj][i] + W[ii][jj] + W[ii][i]; \]
\[ \text{count}--; \]
else
{
\{ if ((Edge[ii][i] == 0) && (Edge[jj][i] == 1))
\{ 
   Edge[jj][i] = 0;
   count--; 
\}
else
if ((Edge[ii][i] == 1) && (Edge[jj][i] == 0))
\{ 
   Edge[ii][i] = 0;
   count--; 
\}
\}
Edge[ii][jj] = 0;
count--; 
Tag[ii] = Tag[jj];
}
B3. CONVERT OUTPUT FILE

The Partitioned Dependence Matrix uses the same output and input number as in the Dependence Matrix input file.

The general format of the Partitioned Dependence Matrix is

<Leader> : <Listing>

where

<Leader> is an input,

<Listing> listing of inputs that should be connected to the leader input during test. If there is no input that can be connected with the leader input, the leader is left alone.

As an example, the Partitioned Dependence Matrix file c17.out, corresponding to c17.w is given below:

```
c17.out
(1) :
(2) :
(3) :
(4) : 0
```

In this example inputs 1, 2 and 3 should be stand alone. Inputs 0 and 4 should be connected together, i.e., they belong to the same set.
In this Appendix proof of theorem 4.1 is presented. According to this theorem, an \((n,k)\)LFSR is capable of pseudo-exhaustive testing an \((n,w)\)CUT, where \(w \leq k \leq n\).

**Theorem 4.1**

An \((n,k)\)LFSR is capable of pseudo-exhaustively testing of an \((n,w)\)CUT if \(k\) is the smallest integer such that

\[
 w = \left\lfloor \frac{k}{n-k+1} \right\rfloor + \left\lceil \frac{k}{n-k+1} \right\rceil \tag{4.1}
\]

where \(\lfloor \alpha \rfloor\) is the greatest integer less than or equal to \(\alpha\), and \(\lceil \alpha \rceil\) is the smallest integer larger than or equal to \(\alpha\).

**Proof**

Note that an example is carried out along the proof of this theorem.

Theorems 2.1 and 2.2 prove that an \((n,k)\)LFSR with \(f(x) = g(x) \cdot p(x)\) has a period \(T = 2^k - 1\), if \(p(x)\) is primitive of degree \(k\), \(g(x)\) is any polynomial of degree \(n-k\), and \(S_0(x)\) is divisible by \(g(x)\). It remains necessary to find the smallest \(k\) such that all \(2^w\) (from any stages of the LFSR) test patterns are generated.

From coding theory, a \(k\)-by-\(n\) generator matrix \(G\) \((k<n)\) can be formed form

\[ g(x) = g_0 + g_1 x + g_2 x^2 + \ldots + g_{n-k} x^{n-k} \]

by shifting the row vector of \(g(x)\) to the right \(k-1\) times, or
which after elementary row manipulation can be represented by

\[ G' = [I : B] \]  

(C.2)

where \( I \) is a \( k \)-by-\( k \) identity matrix.

Consider for instance, an \((n,k)=(10,8)\)

\[
\begin{bmatrix}
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
\end{bmatrix}
\]  

(C.3)

and

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\end{bmatrix}
\]  

(C.4)
Where the operations performed on each line are:

\[
\begin{align*}
L_1 &= L_1 + L_2 + L_4 + L_5 + L_7 + L_8 \\
L_2 &= L_2 + L_3 + L_5 + L_6 + L_8 \\
L_3 &= L_3 + L_4 + L_6 + L_7 \\
L_4 &= L_4 + L_5 + L_7 + L_8 \\
L_5 &= L_5 + L_6 + L_8 \\
L_6 &= L_6 + L_7 \\
L_7 &= L_7 + L_8 \\
L_8 &= L_8
\end{align*}
\]

It can be seen that matrix $G$ has rank $k$, and the first $k$ columns are linearly independent. By row manipulation, it could also be obtained a $k$-by-$k$ identity matrix in the last columns of $G$. This means that, if $n-k<k$, the last $n-k$ columns of $G$ are linearly independent. Also, matrix $B$ describes the dependence relations between the last $n-k$ and the first $k$ columns.

Matrix $B$ is of the form

\[
B = \begin{bmatrix}
V \\
U \\
U \\
U \\
U
\end{bmatrix}
\]  

where $V$ is a $m$-by-$(n-k)$ matrix, in which

\[
m = k \ mod \ (n-k+1)
\]

and $U$ is a $(n-k+1)$-by-$(n-k)$ matrix.

Matrix $U$ is composed of an $(n-k)$-by-$(n-k)$ identity matrix and an all-ONE row vector, and matrix $V$ corresponds to the last $m$ rows of matrix $U$. Consider for
instance, from (C.4)

\[
U = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}
\]  \hspace{1cm} (C.6)

and

\[
V = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}
\]  \hspace{1cm} (C.7)

There is a total of \( k/(n-k+1) \) matrices \( U \) in matrix \( B \), and thus, the minimum column weight of matrix \( B \) is

\[
W(P) = 2 \left( \frac{k}{n-k+1} + \frac{k}{n-k+1} - 1 \right)
\]  \hspace{1cm} (C.8)

or simply

\[
W(P) = \left\lfloor \frac{k}{n-k+1} \right\rfloor + \left\lfloor \frac{k}{n-k+1} \right\rfloor 
\]  \hspace{1cm} (C.9)

which means that each column (input variable) on the last \( n-k \) columns depend on at least \( W(P) \) columns of the first \( k \) columns.

On \( G' \), the identity matrix \( I \) was formed on the first \( k \) columns, but it could be formed on any position. Since the identity matrix \( I \) can be present in any position, (C.9) is always true. The proof of this theorem is based on [Wang85b].
A subroutine to simulate the LFSR have been developed (in C programming language) and implemented into the original fsim. The fsim was not able to simulate an LFSR as test pattern generator. The original fsim was able to read test patterns from an input file, or to use random pattern generator. The number of test patterns used in the Reduced Test Methods could not read the files from an input file given a prohibitively large file that would be required.

The subroutine implemented uses the LSFR model given in Figure 2.3 Chapter 2), represented by equation (2.3) and characteristic polynomial given by equation (2.7).

The modified version of fsim, called fsimm.ou, incorporates the LFSR simulation capabilities. fsimm.ou is able to generate a test pattern and fault simulate the CUT using the test pattern generated. The program performs this procedure until the desired number of faults have been detected, or a limit in the number of test patterns is reached.

The listing of the subroutine that simulates the LFSR is given bellow.
int pget_test(first, input, npi, nbit)
level input[];
int first, npi, nbit;
{
    static F[250], P[250], lfsr[250], clock;
    register int i, j;
    char c;
    int ntest, k;
    unsigned mask1=-(ALL1<<1);
    /* Obtain number of k */
    if (first==1)
    {
        printf("The circuit has %d inputs.\n", npi);
        printf("What is the degree of the generator polynomial? -- > ");
        scanf("%d", &k);
        if (k>npi) k=npi;
    }
    /* Reset of the working polynomials */
    for (i=0; i<=npi; i++)
    { 
        P[i]=0;
        F[i]=0;
        lfsr[i]=0;
    }
    /* Reading of p(x) */
    printf("Please input a polynomial of degree %d.\n", k);
    printf("Input the exponents of the terms different of zero.\n");
    scanf("%d", &c);
    c=0;
    for (; c!='\n';)
    { 
        c =getchar();
        if (c>='0' && c<='9')
        {
            ungetc (c, stdin);
            scanf("%d", &i);
            P[i]=1;
        }
    }
    /* Reading of h(x) */
    if (k<npi)
    {
        printf("Please input a monic polynomial of degree
%d." , npi-k);
        printf("Input the exponents of the terms different of zero.\n");
        c=0;
        for (; c!='\n';)
        { 
            c =getchar();
            if (c>='0' && c<='9')
ungetc (c, stdin);
scanf("%d", &i);
lfsr[i] = 1;
}
/* Obtaining \( f(x) = h(x) \cdot g(x) \) */
for (i = 0; i <= k; i++)
    if (P[i] == 1)
        for (j = 0; j <= (npi - k); j++)
            F[i + j] = (F[i + j] ^ lfsr[j]) & 1;
else
    { /* Reading the seed */
        printf("Please input the seed of the LFSR (degree %d) \n", k);
        printf("Input the exponents of the terms different of zero. \n");
        c = 0;
        for (; c != '\n';)
        {
            c = getchar();
            if (c >= '0' && c <= '9')
            {
                ungetc (c, stdin);
                scanf("%d", &i);
lfsr[i] = 1;
            }
        }
    }
/* Taking the number of clock cycles */
printf("How many clock cycles? ---> ");
scanf("%d", &clock);
if (clock <= 0)
    return (0);
for (ntest = 0; ntest < nbit; ntest++)
{
    for (i = 0; i < npi; i++)
    {
        if (lfsr[i] == 0)
            input[i] <<= 1;
        if (lfsr[i] == 1)
        {
            input[i] <<= 1;
            input[i] |= mask1;
        }
    }
    j = lfsr[npi - 1];
    for (i = npi - 1; i > 0; i--)
        lfsr[i] = lfsr[i - 1] ^ (F[i] & j);
lfsr[0]=j;
clock--;
if (clock==0)
    break;
}
return (ntest);
D2. fsimm.ou INPUT DATA

The fsimm program can be run as follows

\texttt{fsimm.ou <Options> <name.isc>}

where

\texttt{<Options>}  -t if the LFSR simulation subroutine is to be used

- \texttt{r} if the random patterns generator is to be used,

\texttt{<name.isc>}  is the name of the circuit file (ISCAS85 format),

If the option \texttt{-t} is used, the program will ask some information regarding the input circuit and regarding the characteristics of the respective LFSR. Consider the example below, where the underline refers to information supplied (typed) as the program runs.

\texttt{fsimm -t c880.isc}

What is the number of undetectable faults? \text{"\texttt{--\textgreater 0}"

The circuit has 60 inputs.
What is the degree of the generator polynomial? \text{"\texttt{--\textgreater 59}"

Please input a polynomial of degree 59.
Input the exponents of those terms different of zero.
\begin{verbatim}
  0 37 38 58 59
\end{verbatim}

Please input a monic polynomial of degree 1.
Input the exponents of those terms different of zero.
\begin{verbatim}
  0 1
\end{verbatim}

How many clock cycles? \text{"\texttt{--\textgreater 10000}"

\texttt{Q}
The fsimm (and the original fsim) provided as output information such as the circuit structure, simulation parameters, simulation results and CPU time.

In the circuit structure section, depth of the circuit refers to the number of levels if the circuit is levelized. Levelization refers to assigning levels to the circuit nodes, in which a gate output always depend on inputs of a previous level.

In the simulation results section, number of collapsed faults refers to the effective total number of faults. Since two or more different faults in different places may be detected by the same test pattern, these faults can be collapsed to a single fault.

The output data of the fsimm applied to the c880.isc is given bellow.
End of fault simulation.

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

1. Circuit structure
   Name of circuit : c880.isc
   Number of gates : 443
   Number of primary inputs : 60
   Number of primary outputs : 26
   Depth of the circuit : 25

2. Simulator input parameters
   Simulation mode : LFSR

3. Simulation results
   Number of test patterns applied : 5120
   Fault coverage : 100.000 %
   Number of collapsed faults : 942
   Number of detected faults : 942
   Number of undetected faults : 0

4. CPU time
   Initialization : 1.283 secs
   Fault simulation : 4.317 secs
   Total : 5.600 secs
Testing of Integrated Circuits (ICs) is becoming long, complex and expensive, reaching up to a third of the IC price. The testing can be simplified by placing additional circuitry inside the IC to perform the test, known as Built-In Self Test (BIST). The testing time can be reduced using pseudo-exhaustive techniques, although it still remains long.

This work is dedicated to investigate the pseudo-exhaustive testing techniques, and development of the Reduced Test Methods, for the BIST environment. In the investigation of the pseudo-exhaustive testing methods, new algorithmic approaches have been developed and implemented in the C programming language.

The Reduced Test Methods are developed, based on pseudo-exhaustive testing techniques. The Reduced Test Methods determine the minimum number of consecutive test patterns to be applied to a Circuit Under Test (CUT) to detect all faults. The test patterns are obtained from a Linear Feedback Shift Register (LFSR), which are designed based on the pseudo-exhaustive testing techniques. The minimum number of test patterns is obtained from fault simulation of the CUT and simulation of the LFSR. For each test pattern generated by the LFSR, the CUT is fault simulated, and the detected faults are marked. This procedure is repeated until all faults are marked. The total number of test patterns necessary to mark all faults, obtained from fault simulation, corresponds to the total number of clock cycles applied to the actual circuit in order to detect all faults. The fault simulation is performed for different LFSRs, and the LFSR candidate resulting in the shortest