uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Junfeng Li
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.Sc. (Systems Sciences)
GRADE / DEGREE

Department of Systems Science
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Improved Test Efficiency in IP Cores Using ModelSim Verification Tool

TITRE DE LA THÈSE / TITLE OF THESIS

Dr. Amiya Nayak
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

Dr. Sunil Das
CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Dr. Miodrag

Dr. Voicu Groza

Gary W. Slater
Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Improved Test Efficiency in IP Cores Using ModelSim Verification Tool

Junfeng Li

Thesis submitted to the

Faculty of Graduate and Postdoctoral Studies

In partial fulfillment of the requirements

For the M.Sc degree in Systems Science

University of Ottawa

Ottawa, Ontario, Canada

December 2007

© Junfeng Li, Ottawa, Canada, 2008

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-50899-2
Our file Notre référence
ISBN: 978-0-494-50899-2

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Acknowledgment

I would like to express my appreciation to my supervisor, Dr. Sunil R. Das, Professor Emeritus, School of Information Technology and Engineering, University of Ottawa, Ottawa, Ontario, Canada, for his patient guidance, support, comments and encouragement during the entire period of this research. Without him I would never be able to finish this thesis.

I also would like to sincerely thank my co-supervisor, Dr. Amiya Nayak, Professor of School of Information Technology and Engineering, University of Ottawa, for this constant help during this research. His advice greatly improves the quality of this thesis.

Thanks are also due to Mr. Altaf Hossain, for his unforgettable help with my research work.

Finally, I would like to give my truest thanks to my family for never-ending love and support.

Abstract

The complexity of modern digital circuit has increased enormously particularly in the context of paradigm shift from system-on-board to designs embracing embedded cores-based System-on-Chips (SoCs). This increased complexity of circuits in turn results in a huge challenge of setting up their appropriate fault testing environments. Though lots of efforts have been taken to rapidly test the very large scale integrated (VLSI) circuit chips with very reasonable cost, with advances in technology, new frontiers also emerged. This thesis aims at developing a new technique to verify and test architecture of circuits under hardware and software co-design environment, targeting specifically embedded cores-based systems-on-chip. The well-known concept, design for testability (DFT), is utilized in this thesis based on the use of ModelSim simulation and verification tool to simulate the entire design. Some partial results on ISCAS 85 combinational and ISCAS 89 sequential benchmark circuits are provided along with a comparison of the results from some earlier works.

Table of Contents

Abstract iii

List of Figures vii

List of Tables ix

Chapter 1 Introduction 1

1.1 Integrated Circuits 1

1.2 Digital Circuit Testing 2

1.3 Classification of Tests 4

1.3.1 Use of Testing 4

1.3.2 Testing Technology Used 5

1.4 Digital Circuit Testing Problems 6

1.5 Motivation 7

1.6 Thesis Contribution 8

1.7 Organization of the Thesis 8

Chapter 2 Automatic Test Pattern Generation and Fault Simulation 10

2.1 Exhaustive Test Pattern Generation 10

2.2 Pseudorandom Test Pattern Generation 11

2.3 Deterministic Test Pattern Generation 15

2.4 Fault Models 17

2.4.1 Behavioural Fault Model 18

2.4.2 Functional Fault Model 19

2.4.3 Structural Fault Model 20

2.4.4 Switch-Level Fault Model 21

2.4.5 Geometric Fault Model 23

2.4.6 Fault Equivalence and Dominance 24

2.5 Fault Simulation..24
2.5.1 Parallel Fault Simulation..25
2.5.2 Concurrent Fault Simulation..27
Chapter 3 Fault Injection..29
3.1 Fault Injection..29
3.1.1 Fault Injection Overview..29
3.1.2 Dependability Measurement..29
3.1.3 Fault Injection Techniques..30
3.1.3.1 Fault Injection Environment..30
3.1.3.2 Injection Approaches..32
3.1.3.3 Hardware-based Fault Injection..32
3.1.3.4 Software-based Fault Injection..34
3.2 Fault Injection to Verilog Circuit Model..35
Chapter 4 Fault Injection Under Software and Hardware Co-design Environment..36
4.1 Hardware Design Environment..36
4.1.1 Introduction to Verilog HDL..36
4.1.2 VHDL..37
4.1.3 Verilog HDL..37
4.1.4 Testbench..39
4.1.5 ModelSim with PLI Design Environment..39
4.2 Hardware and Software Co-design Framework..42
Chapter 5 Design and Implementation of PLI Fault Injection for Combinational Circuits..46
5.1 General Structure of Combinational Circuits..46
5.2 Design of PLI Fault Injection..47
5.2.1 Overview of Design of PLI Fault Injection..47

5.2.2 Initialization of Fault Injection 51

5.2.3 Random Test Pattern Generation 51

5.2.4 Fault List Generation 53

5.2.5 Fault Injection 53

5.2.6 Comparator Unit 56

5.2.7 Testbench Design 56

Chapter 6 Design and Implementation of PLI Fault Injection for Sequential Circuits 58

6.1 General Structure of Sequential Circuits 58

6.2 Design of PLI Fault Injection 58

6.2.1 Overview of Design of PLI Fault Injection 58

6.2.2 Initialization of Fault Injection 60

6.2.3 Random Number Generation 60

6.2.4 Fault List Generation 64

6.2.5 Fault Injection 65

6.2.6 Comparator Unit 68

6.2.7 Testbench Design 69

Chapter 7 Test Case and Experimental Results 72

7.1 Pseudorandom Test Pattern Generation 72

7.2 Fault list generation 73

7.3 Fault injection simulation 73

7.4 Test Results and Analysis of Combinational Circuits 75

7.5 Test Results and Analysis of Sequential Circuits 78

Chapter 8 Conclusions 82

References 84

List of Figures

Figure 1.1 Rule-of-ten......3

Figure 1.2 General digital circuit testing......4

Figure 2.1 Fault coverage vs. number of test patterns......12

Figure 2.2 A typical block diagram of BIST......13

Figure 2.3 Standard LFSR......14

Figure 2.4 Behavioural description......18

Figure 2.5 Functional description......19

Figure 2.6 Structural description......20

Figure 2.7 Stuck-at fault model......21

Figure 2.8 Switch-level description......22

Figure 2.9 Geometric description......23

Figure 2.10 Parallel fault simulation......26

Figure 3.1 Example of a fault, an error, and a failure......30

Figure 3.2 Fault injection environment......31

Figure 4.1 Netlist of C17......38

Figure 4.2 General view of testbench module......39

Figure 4.3 Typical PLI working procedure......42

Figure 4.4 top-down design flow......43

Figure 4.5 Design flow of fault simulator......45

Figure 5.1 Design flow of PLI fault injection for combinational circuits......47

Figure 5.2 Structure of PLI fault injection......50

Figure 5.3 Linear Feedback Shift Register......51

Figure 5.4 Hardware fault injection scheme......54

Figure 6.1 Design flow of PLI fault injection for sequential circuits......59

Figure 6.2 Structure of PLI fault injection........61
Figure 6.3 An example circuit........62
Figure 6.4 An example circuit with SA0 ant Line F........67
Figure 7.1 Fault coverage of ISCAS85 benchmark circuits by PLI approach........76
Figure 7.2 Detected faults vs. total faults, and test patterns........77
Figure 7.3 Fault coverage of ISCAS 89 benchmark circuits by PLI........79
Figure 7.4 Number of faults simulated vs. number of detected faults and test patterns.....80
Figure 7.5 Number of inputs vs. fault coverage in each sequential circuit........81

List of Tables

Table 2.1 Definition of D algorithm....16

Table 3.1 Fault injection implementation by fault models....32

Table 7.1 Combinational circuit description....75

Table 7.2 Results of fault injection by PLI approach....76

Table 7.3 Results comparison....78

Table 7.4 Sequential circuit description....79

Table 7.5 Results of fault coverage by PLI fault injection....80

Chapter 1

Introduction

Electronic system and digital circuits are permeating our daily life. Due to the rapidly increased demands of our social activities, each circuit will be assigned more and more tasks that need to be finished as soon as possible. The current solution to enhance the capability of processing multiple tasks is to increase the complexity and density of digital circuits, well known as System-on-Chip (SoC). These super chips will be the core technology for the next generation multimedia devices, computer chipsets, mobile devices, and robotics. The SoCs are physically integrated by digital, analog and memory circuits on a single cost-effective silicon chip [1]. Each SoC combines with numerous functionally different blocks that provide partial functionality for this SoC.

The testing of digital circuits is a new established research area. It offers a sort of methodologies to detect faults on the digital circuit board. Because of this growth of needs of complexity and density of digital circuits, a better and more effective method of testing digital systems therefore has to be designed to test the reliability of complex digital circuits. The purpose of testing digital circuits is to detect or discover the physical faults or defects produced during the manufacturing processes in order to ensure the quality of the product. The general testing methodology recently used is to manually produce the fault in the digital circuit, apply a set of input test patterns, also known as test vectors, to the digital circuit, and observe the output whether it is different from the fault-free output. The cost of testing digital circuit is mostly concerned. The cost is caused by the time to generate a set of proper test patterns to achieve 100% fault coverage.

1.1 Integrated Circuits

Integrated circuits enable the technology for innovative devices to change the way we live. The integrated circuit allows us to have smaller devices that have more functions and consume less power. It also enables the engineer to put more and more transistors into a single system. Moreover, the integrated circuit is much easier to be designed and more

reliable than the discrete systems. This further makes the manufactory more flexible to test and maintain the product, compared with the discrete systems.

On the other hand, those demands of sophisticated applications push the design of integrated circuit to a higher level of complexity. We can therefore build multiple-purpose system to satisfy various requirements from customer.

The integrated circuit has three key characteristic rather than discrete systems. They are [1]:

1) **Size**. The scalability of the integrated circuit is very large but the physical size is rather small. Millions transistors and wires are placed in a single integrated circuit. This large scalability leads the high performance with low power consumption, due to the smaller resistances.
2) **Speed**. The signals can be switched between 1 and 0 much faster within a chip than they can between different chips in discrete systems. The communication within a chip is also quicker than the communication between different chips. The reason the integrated circuit has higher speed is because of the smaller capacitances the smaller components and wires having on a chip.
3) **Power consumption**. Due to the large scalability and small size of the integrated circuit, the activities of components and wires can consume less power to be driven.

These advantages of integrated circuits are eventually translated to reduce the cost and size of the product. They hence deeply influence our daily life. More and more portable and handheld devices are appearing in the market in this decade.

1.2 Digital Circuit Testing

The complexity of modern digital systems is putting new demands on system testing. The goal of testing is to ensure that there is no any kind of defect occurring in this digital circuit before it goes into market. If this testing fails in any forms, the manufactory will have no choice but recall the product. The cost will be much higher than that of testing. Therefore, testing digital circuit is well noticed and becomes a critical part of the manufacturing process for the digital circuit during product phases.

We can divide the lifetime of a product into four phases [2]: component manufacture (chips and bare boards), board manufacture, system manufacture, and working life of a product (after entering market). The quality of a product in each phase has to be ensured. The rule-of-ten [3], shown in Fig. 1.1, indicates the relationship between the cost of the test and the cost of repair.

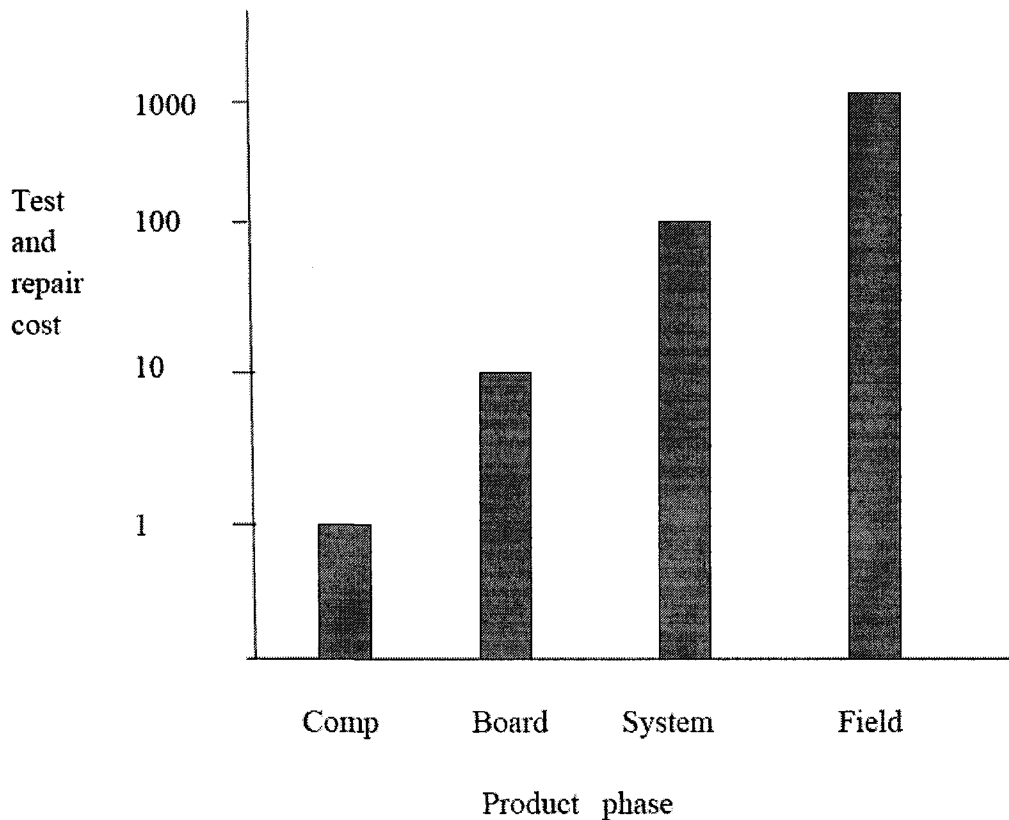

Figure 1.1 Rule-of-ten

The Fig. 1.1 above is showing a fact that, if test and repair cost in component manufacture phase is R, then in Board manufacture becomes 10R, in System manufacture becomes 100R, and in Field phase becomes 1000R. It is obvious that if we can eliminate the fault in the earlier phase, we will have a great saving. Furthermore, a good test methodology will reduce the development time. It can shorten the time between the product is designed and it goes into the market. However, developing test approach also takes time. It is counted in development time so that the development time is increased. Therefore, there is a tradeoff

between test quality and development time to market. If we want to have higher test quality, the development time to market will be extended. In [2], it concludes that the test quality finally applied is not the highest test quality for the reason of total cost.

The Fig. 1.2 below describes a general design of digital circuit testing. This architecture consists of the test pattern generator (TPG), circuit under test (CUT), and the comparator. TPG functions to generate the test patterns to feed the CUT. The comparator compares the output from CUT with the reference stored. A fault is reported if the output does not match the given reference.

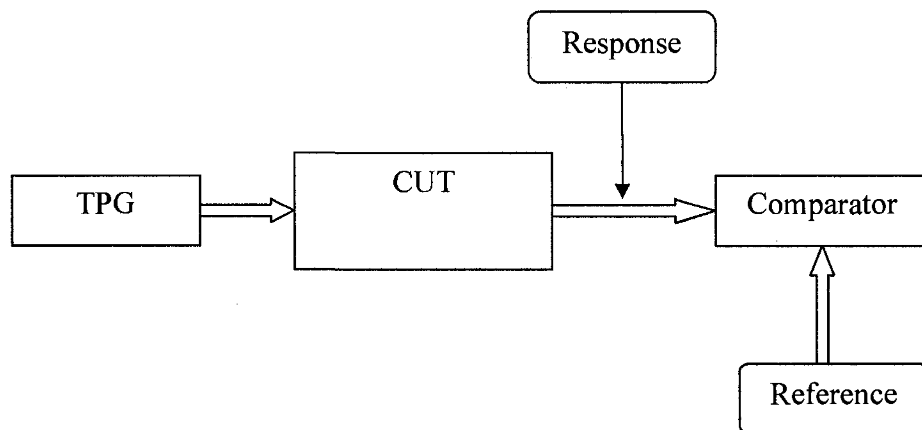

Figure 1.2 General digital circuit testing

1.3 Classification of Tests

A test is a process to distinguish which circuit is good and which one is bad. It can be classified based on the purpose for which the tests are used, or the technology they are using [2].

1.3.1 Use of Testing

Concurrent tests: The testing can be used during the system running. This testing is considered as a part of the system. Such tests are called concurrent tests. These tests are normally applied to correct the coding information. For instance, in parity code concepts,

parity bit is added at the end of the information code to detect single bit errors. In this thesis, this testing technology will not be focused.

Non-concurrent tests: The testing cannot be performed during the system running. It is not a component of the system. It is actually seen as an external part that is used to test the target system by applying various testing approaches. Thus, it is more capable of detecting or locating complex faults. Such testing technology is our focus in this thesis. We will have a deeper looking at this testing technology.

1.3.2 Testing Technology Used

The type of tests is selected depending highly on the circuit to be tested. The circuit is generally divided into three types: analog, digital, and mixed-signal circuit.

Analog circuit: its main property is that the input and output signals are analog. The signals are determined by the supply voltage in a range of 0 to +5V. This analog circuit testing is to determine a range of values of analog parameters such as supply voltage, bandwidth, and frequency response, instead of looking for precise values of these analog parameters due to the nature of analog signals. These analog parameters determined indicate a specification of this analog circuit to ensure it is not damaged in use.

Digital circuit: its main property is that the input and output signals are binary, also well-known as digital. This digital circuit testing is to determine the values of the circuit test response signals by applying a set of digital test patterns into the tested circuit. The digital circuit testing can be precisely achieved due to the nature of digital signals. This thesis will mainly focus on this digital circuit testing.

Mixed-signal circuit: its main property combines analog signal with digital signal. Its input signal is digital (analog) while its output is analog (digital). Digital-to-analog converter circuit is a good given example. Testing mixed-signal circuits require the composition of testing analog and digital circuit techniques.

1.4 Digital Circuit Testing Problems

The first digital circuit test method appeared in early 1960's for Small Scale IC (SSI) [3]. The architecture of SSI was very simple. They just had single gate complexity. Therefore, the test was very easy. Faulty output could be observed in the output and compared with a desired truth-table. Simple equipment such as voltmeter was sufficient enough to finish this test.

With the growth of IC size, the architecture of IC is becoming more complicated. Hundreds of thousands or even more logic gates are existing in one small IC. The number of required test patterns has been increased. The traditional test method could not afford this large number of test patterns. Its weakness came up. The problems began to appear with the weakness of the test method. The problems are classified to input combinational problem, the sequential problem, the gate-to-pin ratio problem, the test pattern generation problem, and the fault simulation problem [3].

1) The input combinational problem. A combinational circuit with N primary inputs has a total set of 2^N potential test patterns, which is able to give this circuit exhaustive testing to examine all the functions of this circuit. For example, a normal 32-bit combinational circuit requires 2^{32} test patterns for an exhaustive test. If we assume each test cycle takes 0.5 sec, the complete test will take around 68 years to finish all functional tests. This time cost is absolutely unacceptable. The problem here arises. If we cannot generate all test patterns for the to-be-tested combinational circuit, what technique do we use to generate the finite number of test patterns to detect faults?

2) The sequential problem. The sequential circuit is the one whose next state is determined by current state. It contains the combinational circuit's feature, plus a number of flip-flops connected to the combinational circuit. This architecture therefore increases the number of test patterns to exhaustively test the whole sequential circuit. The number of test patterns must consider the number of states of the circuit, and consequently it becomes very larger number. The sequential circuit testing still has time and economic problem.

3) The gate-to-pin ratio problem. SSI was easy to be tested not only because just a single gate was involved, but also most of nodes under test were accessible for the primary input or output. But in modern VLSI, a lot of internal nodes inside the circuit are too deep to be controlled by the signal and observed in the primary outputs. The recent VLSI design just increases the number of gates inside the circuit without increasing the number of I/O pins. As a result, several clock cycles are needed to activate the circuits and observe the signal in primary output.

4) Test generation and fault simulation problem. The exhaustive test set for SSI could be created manually. But the test set for LSI or VLSI has to be automatically created by computer. The increased circuit size has pushed this generation time to weeks or even months. A desirable test generation method must be able to generate more effective test patterns in a reasonable time. The computer algorithms for ATPG is suitable for combinational circuit testing, but not for sequential circuit testing. Sequential circuit demands too much memory since different states have to be considered. The full-scan design is wildly used at present. This design separates the combinational circuit and sequential circuit during the test process. Goel indicated that the number of test patterns for full-scan design increases depending highly on the number of gates inside the circuit [4]. Fault simulation is used to evaluate the test sets in given circuit. It provides the information which fault is detectable in this given circuit. The time consumed in simulation is also increased based on the size of given circuit.

1.5 Motivation

The motivation for this work is to directly test hardware circuit in the Verilog format and improve the test efficiency. In previous works, various fault injection have been proposed, but most of them cannot work on the hardware circuit in the Verilog format. Due to the fact that Verilog HDL is one of the most useful design languages in manufactories for further circuit synthesis in hardware, unable to apply these fault injection approaches to practical circuits becomes a big drawback. A new approach thus has to be created to directly insert the

fault to the hardware circuit in the Verilog format in order to test the circuit performance in practice.

Even though some works has been presented to inject the fault to the circuit in the Verilog format, test efficiency is a primary issue in those works. Testing time becomes very long in complex circuits testing. This is not affordable for circuit development and also unacceptable for manufactories in terms of product cost.

Hence, we are motivated to develop a new approach to both take advantage of fault injection in Verilog hardware circuit and improve its test efficiency to serve the circuit development.

1.6 Thesis Contribution

In this thesis, we focus on using PLI to insert the fault to Verilog hardware circuits in a software and hardware co-design environment. The main contributions of this thesis are the followings:

1. A detailed design and implementation on PLI fault injection approach.
2. Experimentations with ISCAS benchmark circuits based on PLI fault injection approach.
3. Comparison of results of our approach with relative works

1.7 Organization of the Thesis

Chapter 1 gives an overview of the conventional test techniques, concepts, classification of testing, circuit testing problem, and general description of test generation techniques.

Chapter 2 introduces the details on Automatic Test Pattern Generation techniques that are mainly used for fault simulation, followed by detailed introduction of different fault models on each level of circuit design, and its relative fault simulation

Chapter 3 gives the fault injection approaches that are representative of the important work in this research field.

Chapter 4 gives the general overview of hardware description language, especially Verilog VHDL, and the developing environment for circuit testing, ModelSim.

Chapter 5 and Chapter 6 focus on detailed design and implementation of the new fault simulation method for combinational circuit and sequential circuit.

Chapter 7 presents test case and experimental results of combinational and sequential circuits, and

Chapter 8 summarizes the whole work, general conclusions of this new fault simulation method, and describes future trend in this research field.

Chapter 2

Automatic Test Pattern Generation and Fault Simulation

Test pattern generation plays very critical role in the fault simulation. It functions to generate test patterns to feed the CUT during the test generation process. The test pattern can be produced either by test engineers specifically for the purpose of exercising the targeted faults in the circuit, or by an automatic test pattern generation (ATPG).

Test generation approaches can be generally classified into three categories: exhaustive, pseudorandom, and deterministic. For the combinational circuits with the small number of primary inputs, exhaustive test generation is the best candidate, because this generation method is capable of producing all possible test patterns that guarantees to detect all detectable faults. Some efforts have been given to extend these exhaustive techniques to larger circuits by partitioning these large circuits to many subcircuits where exhaustive generation can be applied to each subcircuit. However, finding suitable partition is neither easy nor guaranteed [10].

2.1 Exhaustive Test Pattern Generation

Exhaustive test pattern generation is the most accurate algorithm to test all the testable faults in a digital circuit. Exhaustive test pattern generation can generate all the possible test vectors, whose total number is 2^N, where N is the number of inputs of the digital circuit. For example, for a 2 input digital circuit, there are 2^2 test patterns needed to exhaustively test the digital circuit, while there are 2^{20} test patterns needed to do the exhaustive testing for a 20 input digital circuit.

In terms of cost of testing, the exhaustive test pattern generation will take more and more time on generating the whole test vectors, when the number of inputs is being increased, which is normally happening in modern digital circuit design. As a result, the exhaustive test

pattern generation is generally applied in small-scale digital circuit testing. For large-scale digital circuit testing, a new approach is needed.

2.2 Pseudorandom Test Pattern Generation

Random test pattern generation offers a simple and inexpensive way to detect faults in a circuit under test (CUT). Although it is hard to guarantee for high fault coverage, especially in some cases of testing large circuits, pseudorandom test pattern generation is still very useful in a wide range of practical circuits. The term pseudorandom refers to a manner that describes a fact that the appearance of 0's and 1's sequences are repeatable due to the restriction of hardware and its generation. Manufacturing testing must periodically repeated during the entire lifetime of a chip. Pseudorandom manner provides this opportunity to allow test engineers monitor the chip condition during the period of maintenance. Test pattern generation happens mostly in software domain [28]. Regardless of given test pattern generation algorithms, the properties of resulting test sets are not altered. Fig. 2.1 shows the typical fault coverage vs. number of test patterns curve. If we generate pseudorandom test patterns as many as we can, this method eventually become exhaustive test pattern generation. It is therefore expected to 100% fault coverage. Normally, test engineers apply pseudorandom test pattern generation at the beginning of the test generation process to quickly remove those easily detectable faults from fault list, and then use deterministic ATPG to eliminate those hard detectable faults. This composition of two test generation methods can greatly influence the test cost.

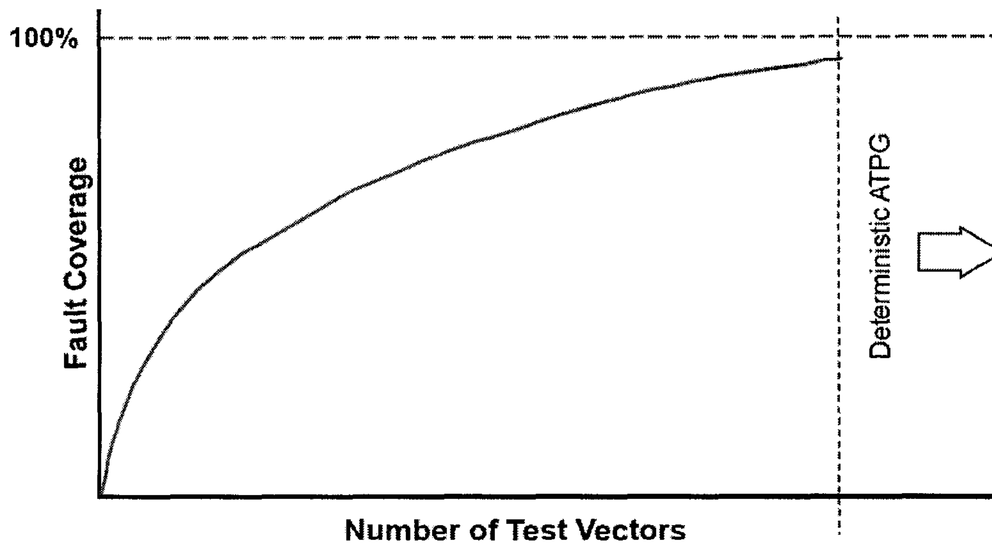

Figure 2.1 Fault coverage vs. number of test patterns.

- Built-In Self-Test

Built-in self-test (BIST) is a technique that enables the chip to test itself. This technique combines with built-in hardware and self-test methodology [5, 48]. Built-in hardware refers to the one incorporated on-chip. This hardware allows that all test generation, test applications, and response verification are accomplished without external power, clock and tester required. It can test different parts of chip in parallel, therefore reducing the test time [6, 44, 51]. Self-test methodology refers to provide a complete circuit testing processes by the chip itself. Self-test can run at the circuit's clock speed so that the test function can keep pace with increasing circuit speed. Since it is integral part of the system, self-test can be trigged anytime on request by another application. This aspect enhances the reliability of the system.

Fig. 2.2 depicts a classic BIST architecture. It contains test pattern generator (TPG) that provides test patterns for the next block. Module under test (MUT) represents the circuit that needs to be tested. Compaction Unit is used to compact the test responses to feed the comparator. Reference signature is the correct responses that are compared with the test responses to find out whether a fault is detected.

The most widespread used BIST schemes for the module are the test-per-scan scheme and the test-per-clock scheme [5]. In test-per-scan scheme, each test vector is firstly shifted into the scan chain, and then applied into MUT. After m+1 clock cycle, where m is the number of flip-flop in a scan chain, the response of MUT is loaded back to scan chain. The content in current scan chain is compared with the fault-free output previously captured in the comparator unit. The test-per-clock, on the other hand, generates test patterns, tests MUT, captures response and compares response with fault-free output in each clock cycle. This test-per-clock, compared with test-per-scan, is able to shorten test times as a new test pattern is always generated in each clock cycle without any delays. However, test-per-clock requires larger test registers in order to handle the test data generated in each clock cycle, which may have critical impact on the circuit performance. Hence, it is very important to be aware of the fact that a proper test pattern generation method and MUT testing approach can achieve high fault coverage and reduce the test cost.

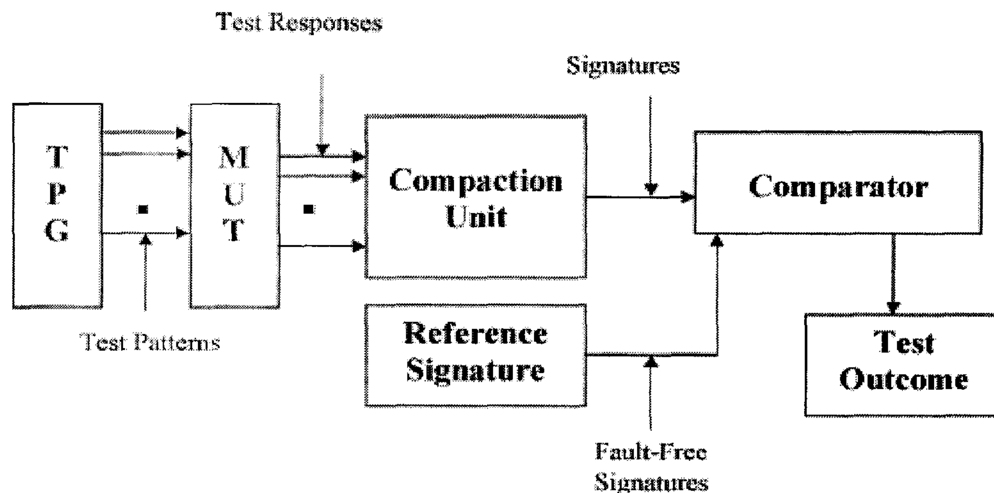

Figure 2.2 A Typical Block Diagram of BIST [6]

- Linear Feedback Shift Registers

A linear feedback shift register (LFSR) is a modified shift register whose input bit is linear function of previous state. Its length is determined by a number of 1-bit memory cells

[2, 42]. This operation requires having a feedback loop to move the last memory element to the first one. This well-chosen feedback function guarantees a sequence of bits which appearance randomly. Once the final state is reached, the LFSR will traverse the sequence exactly as before. The XOR logical gate in most of cases is used to accomplish the LFSR. Fig. 2.3 shows a typical LFSR.

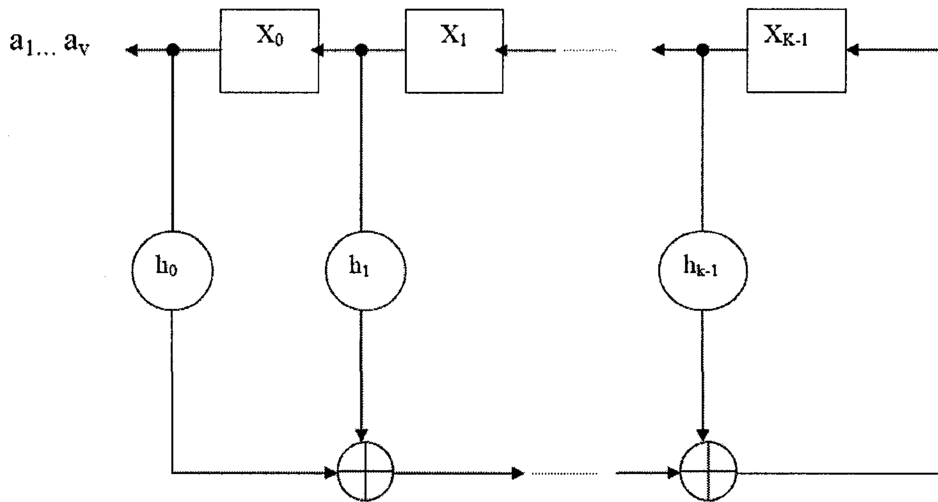

Figure 2.3 standard LFSR

The maximum length of sequence of the LFSR is 2^k-1, where k represents the number of bits the LFSR has. The general characteristic polynomial for LFSR is:

$$h(X) := X^k + h_{k-1}X^{k-1} + \ldots\ldots. + h_1X + h_0$$

Where $h_0, h_{1,},\ldots, h_{k-1}, hj \in \{0, 1\}$ are defined as feedback coefficients that determine the behaviour of a LFSR. It determines the length of the sequence generated by an LFSR. The term tap refers to lines that run from the output of one register within the LFSR into XOR gates that determine input of another register within this LFSR. The tap is chosen based on the characteristic polynomial. Registers that do not need taps can operate as a standard shift register. The chosen taps will determine the number of values in the sequence before repeated. When the feedback function is activated, the selected bit values are collected before the register is clocked. The result of the feedback function is inserted into the shift register during the shift to fill the position that is emptied as a result of the shift through taps.

2.3 Deterministic Test Pattern Generation

Automatic test pattern generation (ATPG) algorithms utilize the gate-level description of a digital circuit to generate a condensed set of test vectors. ATPG algorithms provide a mechanism to generate a test vector for a specific fault, and fault simulation is applied to determine whether any additional faults are also covered by a given vector. ATPG mostly is divided into three steps. In the first step, a fault-independent method is applied to generate a set of test patterns to test most easy-detectable faults, which is a larger percentage of total faults. At the second step, a fault-oriented method is applied in order to generate a specific test pattern for a given not-easy detectable fault. At the last step, an attempt is made to reduce the generated test set. Compared with exhaustive test pattern generation, this ATPG is more efficient as it just generates all the useful test patterns, and eliminates those useless test vectors. As a result, it is possible to test large-scale circuit within a reasonable period of time.

The required number of test patterns is significantly much smaller when deterministic test pattern generation algorithm is used. In order to recognize the fault in the circuit, all modern ATPG algorithms must parse the data structure of the CUT to generate the test pattern. The basic elements of any ATPG search algorithms include four phases [2, 28, 34]:

1) Sensitization

2) Excitation

3) Justification

4) Implication propagation

In order to detect a fault in a circuit, this fault has to be excited first. Then we need sensitization to propagate the fault to primary outputs. In turn, justification is applied to assign or verify suitable values. Finally, implication procedure propagates the obtained value via the path from fault location to primary outputs. If there is no inconsistent or conflicting value occurred in the circuit, the test pattern is obtained.

Based on this general procedure mentioned above, there are some algorithms created for ATPG.

- **D-Algorithm**

The D-algorithm is the first known complete test generation algorithm. It was developed by Roth at IBM in 1966. This algorithm is dealing with the fault effect excitation and fault effect propagation, and it also establishes the local value assignment to find out the test vector for a specific fault in the circuit using gate-level stuck at fault model. In the D-algorithm, the notion D and D' will be mentioned to indicate the different status when the circuit is faulty and the circuit is fault free. The idea behind the D-algorithm is to attempt to propagate D or D' to the primary output by applying a particular test vector.

GC / FC	0	1	X	D	D'
0	0	D	0	0	0
1	D	1	X	D	D'
X	0	X	X	X	X
D	0	D	X	D	0
D'	0	D'	X	0	D'

Table 2.1 Definition of D algorithm

The five-valued logic (0,1,X,D,D') of Table 2.1 is used to describe the behaviour of a circuit, indicating the concept of D algorithm. The D represents logic values on two superimposed circuits. If the good circuit and faulty circuit have the same value, the composed circuit value is either 0 or 1. When they have different values, the composed circuit value is D, which indicates the logic value 1 is from good circuit and 0 from faulty circuit. The D' is presented if the logic value 1 is from faulty circuit and 0 from good circuit. X designates a DON'T CARE value. The process to assign a value to primary output is generally divided into two steps:

Step 1: Indicate the local value for fault excitation. In this step, a D or D' is assigned so that the value of faulty line can be specified.

Step 2: Indicate the fault effect propagation. In this step, the focus is on the propagation of D oriented from fault excitation. The D notion keeps moving to the final primary output, or until it disappears. In this case, D notion disappearing indicates the value of current input is not correct for this fault. A backtrack is needed to a suitable test vector that can drive notion D to the final primary output. The process is repeated until a correct value is assign, if this value exists. Otherwise it is reported this fault does not have corresponding value for testing.

D-algorithm is efficient enough for small circuits to generate a test pattern for a selected fault, which propagates back to primary inputs and forward to primary outputs. However, in circuits that has the large number of fanouts, D-algorithm may encounter serious conflicting when the value is alternative. In the worst case, D-algorithm may require each possible gate on a path from fault site. This results in a huge number of unnecessary gates involved [57]. Therefore, D-algorithm is not suitable for this type of circuits to generate test patterns.

2.4 Fault Models

In order to reduce the pain of test generation complexity, the awareness of modeling the actual defects that may occur in a chip with fault models at high level of abstraction is very necessary. From the fault model, designers or users can predict the behaviour of faulty circuit caused by the particular fault. This process of fault modeling eliminates the useless test patterns and consequently reduces the burden of test. This is possible by the fact that every physical defect occurring in a chip can be mapped to a single fault at higher level. Therefore, fault models become more independent of the technology.

The circuit can be described at various levels of abstraction in the design procedure. This mentioned design procedure enables the designer to completely cover all the design details, including the behaviour of circuit, and arrangement inside the circuit and the transistor selected for each application of circuit. These levels are [2]:

- Behavioural
- Functional

- Structural
- Switch-level
- Geometric

2.4.1 Behavioural Fault Model

A *behavioural description* (shown in Fig. 2.4) of a circuit is given by using hardware description language such as VHDL or Verilog. This description can indicate the data and control flow, which can be seen as the behaviour of the circuit. Moreover, the behavioural description can visualize abstractly the interaction among components of the circuit, and allow designers to start synthesis at this level.

```
While (i>j)
{
counter= counter + 1;
}
```

Figure 2.4 Behavioural description

At this level, the behavioural fault model illustrates the erroneous behaviour of the circuit based on the behavioural specification of the system. It represents the complex failures in VLSI designs. Errors associated with the value of the variables or time parameters are deliberately introduced into the hardware description language that is used to describe the behaviour of the circuit. The results from this faulty circuit can be observed so that the fault of this circuit can be identified [18].

The behavioural models have been derived into eight fault classes, which constitute the primary fault model used in this research field. This behavioural fault model is based on a form of model perturbation, which has been approved that it can provide good fault coverage as gate-level fault model does [10]. The eight faults are classified [21, 22]:

- *Stuck-Then*, which represents a failure of the **if-then-else** construct to ever execute the else statement.

- *Stuck-Else*, which represents a failure of the **if-then-else** construct to ever execute the then statement?
- *Assignment Control*, which represents a failure of the HDL assignment operator to assign a new value to a signal such that A<=new_value will never be executed.
- *Dead Process*, which represents a failure of the statement within a process construct to execute.
- *Dead Clause*, which represents a failure of the HDL CASE construct to execute one of the alternativeness of the statements.
- *Micro-operation*, which represents a failure of an operator to perform its intended function of the HDL.
- *Local Stuck-data*, which represents a failure of a signal or variable object to have the correct value, where a signal will be stuck in one expression of the device model.
- *Global Stuck-data*, which represents a failure of a signal or variable being able to change the value within the device model.

2.4.2 Functional Fault Model

A functional description, shown in Fig. 2.5, is given at the register-transfer level (RTL). At this level, there are register, modules such as adders and multipliers, and interconnected structures such as multiplexers and buses. It can be seen as a result of behavioural synthesis that transforms a behavioural description to RTL circuit.

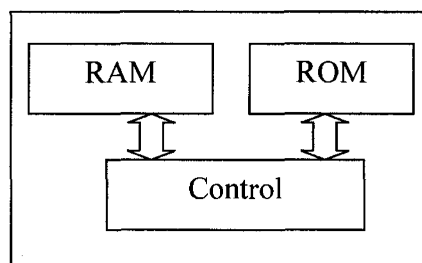

Figure.2.5 Functional description

The functional fault model focuses on the execution of the function blocks of a register or module. It ensures the given functional block performs the function that is designed for, and does not execute any unintended function. The functional fault model contains [26]:

- *Bit-Flip*, which indicates a failure causing a bit changed from 0 to 1 or 1 to 0.
- *Flip-to-1*, which indicates a failure causing a bit always switched to 1, regardless with what value it had.
- *Flip-to-0*, which indicates a failure causing a bit always switched to 0, regardless with what value it had.

The criterion of being a good functional fault model in [2] is that, if a functional fault model is not too complex for test generation, and can result in high fault coverage for the test, this type of functional fault model is considered as a good one.

2.4.3 Structural Fault Model

A structural description, shown in Fig. 2.6, is given at the logic level. It consists of logic gates and interconnected lines among them.

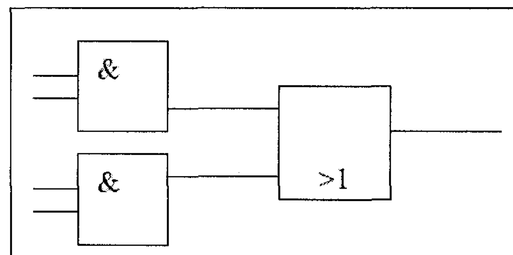

Figure 2.6 Structural description

The structural fault model is dealing with the given structure's capability of carrying both logic 0 and 1. This fault model enables designers to distinguish the faulty circuit and fault-free circuit. The most important fault model at this level is Stuck-at fault model shown in Fig. 2.7. It is assumed that a single line is fixed with logic 0 or 1 by using high or low voltage, respectively. It is said to be single stuck-at fault model if the stuck-at fault occurs only on one line, while it is called multiple stuck-at fault model if the faults simultaneously present on more than one line. The stuck-at fault model corresponds to real faults, although it does not represent all possible faults. It has been proved that the different test programs based on this stuck-at fault model can detect all the faults but not all the faults can be identified [2].

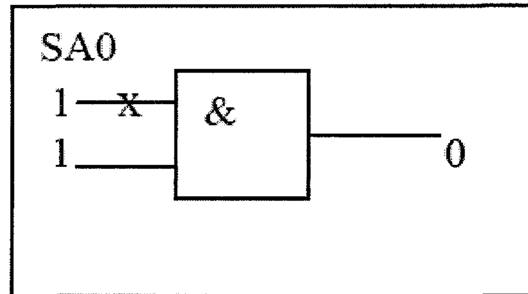

Figure 2.7 Stuck-at fault model

This stuck-at fault model can be specifically divided into three sub fault models in terms of types of gate [26]:

1) *AND gate fault model*: this fault model is for inputs stuck-at-1 and the outputs stuck-at-1 and stuck-at-0.

2) *OR gate fault model*: this fault model is for input stuck-at-0.

3) *Inverter fault model*: this fault model is for input stuck-at-1 and stuck-at-0 and for output stuck-at-0 and stuck-at-1.

The advantages of this stuck-at fault model are that, it can apply not only at logic level, but also at another level. And the algorithm of test patterns generation has been well developed for this fault model. By detecting all or near all stuck-at faults, there is a high probability to detect most physical defects that may occur in a chip. Furthermore, by recording the exact number of faults in a circuit, a fault dictionary can be established for tracking these detected and not detected faults by a test. This information later can be used to create an effectiveness measure or figure of merit for the test.

In this thesis, stuck-at fault model is the only fault model we use in fault injection simulation.

2.4.4 Switch-Level Fault Model

A switch-level description, shown in Fig. 2.8, is given at the transistor-level details of the circuit. In CMOS technology, each logic gate in fact is described by using an

interconnection of a pMOS and nMOS network. In this network, a set of transistor has been interconnected, with 0, 1 and X representing open, close and indeterminate switches, respectively [13]. Furthermore, the transistors are assigned different strengths to model their different resistances in the circuit. Those transistors are normally connected either in series-parallel fashion or non-series-parallel structures.

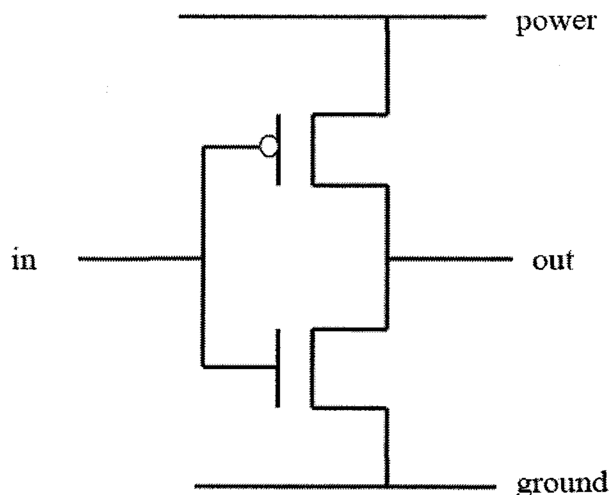

Figure 2.8 Switch-level description

The fault model deals with faults in transistors in a switch-level description of a circuit. The most used fault models at this level are [2,36]:

- The *stuck-open* fault model. This fault model assumes that a transistor or a single physical line is broken, which may result in a permanent non-conducting failure inside the CMOS circuit.
- The *stuck-on* fault model, on other hand, assumes that a transistor or a single physical line is broken, which may result in a permanent conducting failure inside the CMOS circuit.

The advantages of the fault model at this level are that, firstly, they can cover some defects that may not be covered by structural level fault models, and secondly they also can be tested with sequences of stuck at fault tests.

The disadvantage, however, is the increasing number of tests that have to be applied as the CMOS gate can retain its previous logic value at its output in the present of the fault so that two-patterns test, which contains an initialization vector and a test vector, is needed to initialize output, even though it is easy to find out the sequences of tests.

2.4.5 Geometric Fault Model

A geometric description, shown in Fig. 2.9, is given at the layout level. From this description, the line widths, inter-line and inter-component distance, and device geometries are determinate.

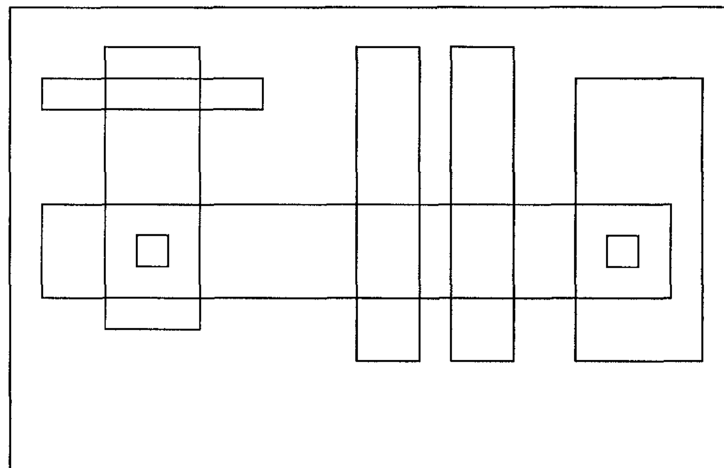

Figure 2.9 Geometric description

The geometric fault models are derived directly from the layout of the circuit. It deals with opens and shorts inside the circuit. The most accurate fault model at this level is bridging fault model. It is more applicable as line widths get smaller. The bridging fault model illustrates the occurrence of shorts between two nodes. Moreover, the bridging fault model sometimes is classified into fault classes [2,37]:

- The fault that causes bridging within logic elements (e.g. transistor gate).
- The fault that causes bridging between logic nodes (e.g. input or output of logic elements) without feedback.
- The fault that causes bridging between logic nodes with feedback.

2.4.6 Fault Equivalence and Dominance

If two faults, f_i and f_j, have identical faulty output, O_f, which means these two faults cannot be distinguished by the test pattern, these two faults are said to be equivalent. The fault dominance has been defined as that, if all test patterns of f_i can also detect f_j, f_i is said to dominate f_j. These two nature features is greatly help reduce the fault simulation complexity, especially fault list size, as the simulation time is highly affected by the fault list size. The reduction of fault size, well-known as fault collapsing, can reduce test generation and fault simulation time. It is more obvious when the complex circuit is simulated.

2.5 Fault Simulation

Simulation normally refers to an experiment through software programs that use models to replicate the behaviour of device and observe the response of this model. For electronic produce, verification of hardware and software is required to assure the quality of products. To directly verify hardware is a costly task. Hence, using software to verify hardware performance becomes very popular due to their economy and accuracy.

The motivation of using fault simulation is to minimize the amount of potential defects in the product before it is shipped to customer. It is true that by improving test techniques, the number of defects can be reduced.

Fault simulation is a process of analysing the performance of circuit under various fault conditions. By comparing the fault simulation results with those of fault-free simulation of the same circuit simulated with the same test patterns, we can determine whether a fault is detected by this test pattern. After applying a serious of test pattern to the circuit, the measurement, fault coverage (FC), is computed. The equation is:

$$FC = \#\ of\ detected\ faults\ /\ \#\ of\ simulated\ faults$$

FC is the standard to measure the quality of test of the fault model. In practice, it is impossible to obtain 100% fault coverage for a VLSI part [2]. The reason is:

1) *The fault model.* Not all actual defects can be mapped to the fault model, or vice versa,

2) *Data dependency of fault.* Some faults can only be detected depending only on the execution of all functions, which may not be sufficient to excise, and

3) *Test limitation.* Some faults cannot be detected because of the limitation, where some parts of the circuit are not accessible.

Fault simulation is mostly performed using gate-level stuck-at fault model due to the fact that it can be mapped to most of real defects, although fault simulation can also be performed using other fault models such as functional fault model. Moreover, stuck-at fault model is easy to automatically inject faults into the circuit model by computer programs. Besides evaluation of stimuli, fault simulation also serves several purposes [34]:

- It confirms detection of a fault for which an ATPG generates test. Element delays occur all the time in all fault simulation. We normally assume that the simulation is ideal enough so that no delays are considered. The fault simulation thus is an effective approach to confirm detections caused by ATPG, not delays.
- It computes fault coverage of specific test patterns. Since ATPG targets specific faults, the ability to identify all detected faults by each test pattern in the fault simulation can reduce the number of iterations through an ATPG, and then further reduce the size of test program.
- It provides ability to diagnose faults. Fault simulation provides results that indicate which test patterns can detect which faults. Knowing these results can help test engineers eliminate the suspect parts and find out the faults on physical circuit board.
- It provides the information of fault coverage for specific parts of circuit. This information gives the test engineer the clue what effort is expected to increase fault coverage in specific part of circuit. Writing appropriate test patterns can boost fault coverage in this area.

2.5.1 Parallel Fault Simulation

Parallel fault simulation is one of the oldest high-performance algorithm for fault simulation and the first algorithm to simulate a number of faulty machines by taking advantage of the word level parallelism of host computer [33]. This allows a good circuit

and N faulty circuits simulated simultaneously, where N = W-1, where W represents word length of host computer. This parallel fault simulation consists of the follow steps:

1) Fault group: all faults in the circuit are partitioned into several fault groups of size N. Faults in the same group are simulated simultaneously.
2) Each nodes has associated with a word with length W. this word contains 1 bit value for good machine and N bits values for faulty machine.
3) Fault injection is performed using bitwise operation to present stuck-at fault model.
4) After a logic gate is evaluated, the logical word operation is applied to identify which fault is detected in this computer word with length W.

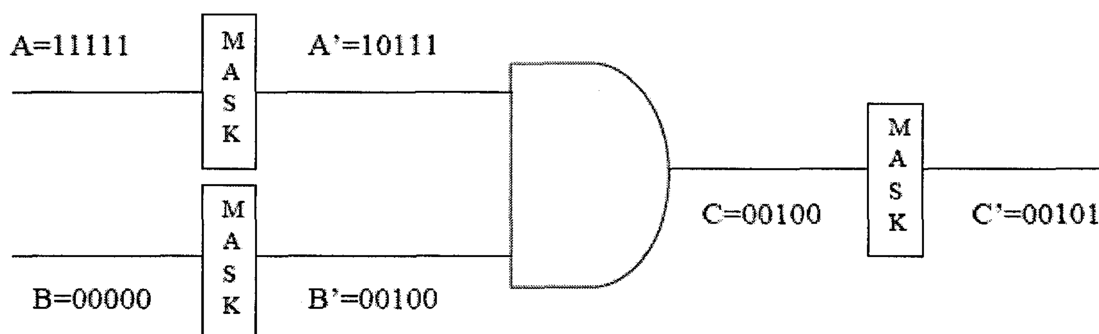

Figure 2.10 Parallel fault simulation

Fig. 2.10 illustrates an example of parallel fault simulation on AND gate when W=5 bits. Suppose we set the 1st bit is fault-free, the 2nd bit is SA0 on line A, the 3rd bit is SA1 on line B, the 4th bit is SA0 on line C, and the 5th bit is SA1 on line C as well. We use MASK as a fault injector to insert faults to each position. On the primary output, C', the 3rd bit and 5th bit with SA1 are detected as the values are different from the 1st bit 0, which is from good machine.

Many fault simulators such as PROOFS [59] and its development HOPE [29, 52, 53, 58] have been developed based on this parallel fault simulation algorithm. The PROOFS fault simulation algorithm is based on performing logic simulation for a particular test pattern to get the state of good circuit. It firstly simulates the fault-free circuit, and then a group of 32

active faults are injected and simulated in parallel. Active fault refers to the fault that causes logical difference on a state line between fault-free and faulty machines in current time frame. If fault is detected, it is dropped from fault list. After a group of 32 faults is simulated, another group of 32 faults is selected and simulated, and then fault list is upgraded again. This process is repeated until all active faults are exhaustedly simulated. In the following, we describe some highlights of PROOFS.

1) Fault ordering and grouping: a proper fault grouping and ordering strategy is crucial to take advantage of parallelism. In PROOFS, fault grouping is based on the results of depth-first search of the circuit from primary outputs to primary inputs in pre-process. Faults that cause same events are grouped in the same fault group. Grouping these active faults results in a significant reduction of the number of simulation.

2) Reduction of fault injection time: the conventional fault injection for parallel fault simulation is to mask the targeted bit of the word [30]. This method requires checking every gate during the fault simulation in order to determine whether the mask is needed. As a result, the whole fault simulation becomes very slow. In PROOFS, a two-input AND gate or OR gate are used, depending on the type of fault, as a fault injector on the faulty line, instead of the bit masking of the word. This method avoids checking each gate and improves the effectiveness of fault injection.

3) Group ID: group id is assigned to each group to distinguish between faulty machine values from different faulty machine groups. According to the faulty value associated with the id, a residual value can be identified and distinguished from previous fault group propagation. It assures the correct value on the line.

2.5.2 Concurrent Fault Simulation

Concurrent fault simulation is essentially a data processing task. In the concurrent fault simulation paradigm [2, 31, 34, 59, 70], the fault-free circuit and the faulty circuit are simulated concurrently. This concurrent fault simulation is inspired from the fact that a fault does not affect the entire circuit and the behaviour of the good and faulty circuits is rarely different. Firstly a fault-free circuit is simulated. Then we just need to simulate faulty part of

circuit that behaves differently. A fault list is built at each node in the circuit. This fault list is reconstructed after resimulation in an event-driven manner, and detected faults are removed from the fault list. Event-driven manner indicates a change in the logic value of a node. Since at each node there is a fault list to store faults, this requires a lot of memory early in the simulation process when many faults are not detected.

Compared with parallel fault simulation, concurrent fault simulation is not as efficient as parallel fault simulation in memory usage as each node has to store undetected faults. Parallel fault simulation is easy to implement. However, as long as the number of faults is large, many simulations have to be performed to simulate all faults. This further results in a significant increase in the computational requirement. The concurrent fault simulation, on the other hand, is faster to deal with a large number of faults due to the fact that it just considers the active fault that change the logical value of the line.

Chapter 3

Fault Injection

3.1 Fault Injection

3.1.1 Fault Injection Overview

Fault injection is a well known technique that is used to introduce controllable faults or errors into the target system. By doing so, it is possible to know how the system behaves in the presence of faults and tests the fault tolerant mechanism in its components or its environments.

3.1.2 Dependability Measurement

Dependability is a system's attribute that quantifies through the specific measure [3]. In general, availability and reliability can be seen as the primary measure in recent research filed. The reliability is defined as a conditional probability of system performing correctly during a specified time interval, while the availability is also a probability of system operating correctly at the instant time [14]. Fig. 3.1 provides an example of the relationship of the fault, error and failure.

Most fault models use fault coverage to measure the probability of successful performing the action which is needed to recover from a fault. Those actions contain detecting faults, identifying faults and recovering faults. The system's workload may impact on the accuracy of the fault model we are taking so that the dependability of the system will be affected; therefore the action has to be taken very quickly, since the accumulation of fault-handling may happen in the system if the action is taken slowly. An approach for this situation is to apply distribution-latency [4], which is the time needed to perform each of these actions. The data from each action can be collected and used to estimate the performance of real system.

Fault injection opens an opportunity to allow us to focus on where, when and how the faults are intentionally inserted as recently the large scalable system is hard to exhaustively insert faults in terms of time constraint. Thus, the investigation only can be done to some

selected faults. The inserted fault must be controllable so that the attributes such as the type, location, time and duration can be known for study purpose. Furthermore, the inserted fault must be the one that can actually affect the system's behaviour; otherwise the fault cannot be observed and distinguished. As well, the system must be operated with a representative workload to obtain a realistic response. Eventually, as the benefit of the fault injection technique, a fault dictionary can be set up to provide the system diagnosis during the period of maintenance, and the understanding of how the systems behave in the presence of faults can be given. This understanding will ultimately lead to better system design and higher dependability.

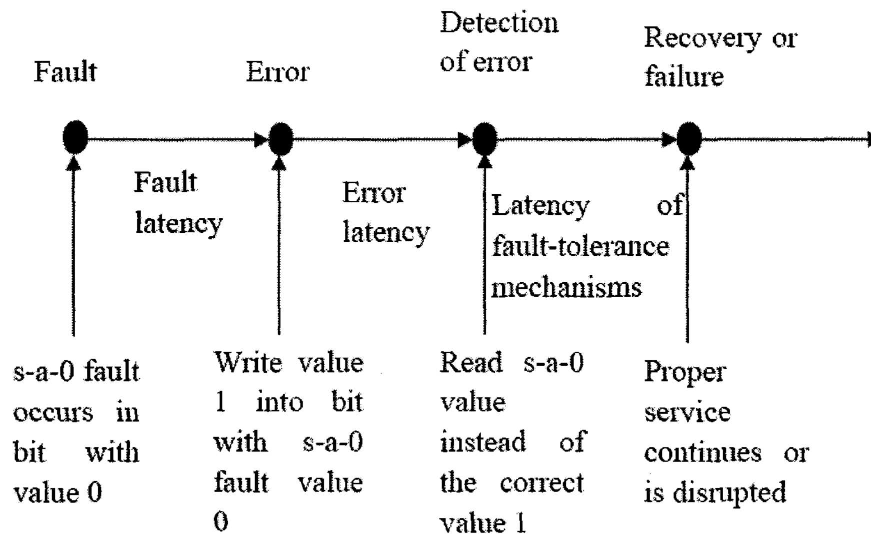

Figure 3.1 Example of a fault, an error, and a failure

3.1.3 Fault Injection Techniques

3.1.3.1 Fault Injection Environment

Currently different fault injection tools and techniques have been developed. Their focus varies depending on the properties of target system. However, they all have the same structure of fault injection environment. Fig. 3.2 depicts a typical fault injection environment. This environment consists of the target system and the fault injection system, which is

constituted by a fault injector, fault library, workload generator, workload library, controller, monitor, data collector and data analyzer [12,17].

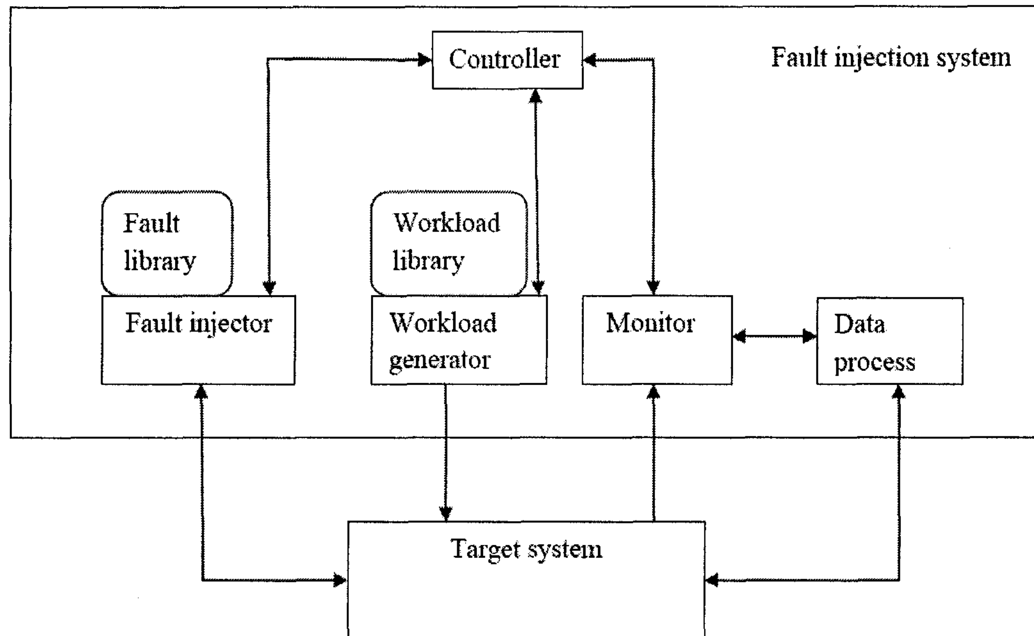

Figure.3.2 Fault injection environment

The fault injector will specify a fault from the fault library and directly inject faults into target system. The workload for the target system is provided by the workload generator in order to simulate the actual workload as close as possible in real system. The monitor is used to track the execution of the commands and initialize the data collector. The data collector performs the real time data collection, while the data analyzer performs data processing and analysis. The controller has the highest priority to control the entire fault injection.

The fault injector can be either hardware implemented or software implemented. The fault injector itself can support different types of fault models, which are stored in the fault library and loaded on request. This structure is flexible and portable as most current fault models can be covered by this single fault injector.

3.1.3.2 Injection Approaches

Generally, fault injection can be divided into two categories, hardware-based fault injection and software-based fault injection. The software-based fault injection requires the fault models which are not required in hardware-based fault injection. Either of these two fault injections can be selected depending on the fault type we are interested in. In some cases, the trade-off exists as some faults will cost more time but higher accuracy for hardware-based fault injection, but less time and less accuracy for software-based fault injection. Table 3.1 shows commonly studied faults and injection [4].

Hardware	Software
Open	Storage data corruption
Bridging	(such as register, memory and disk)
Bit-flip	Communication data corruption
Spurious current	(such as bus and communication network)
Power surge	Manifestation of software defects
Stuck-at	(such as machine level and higher levels)

Table.3.1 Fault injection implementation by fault models.

3.1.3.3 Hardware-based Fault Injection

This type of fault injection applies additional component (hardware) to introduce the fault into the circuit. Depending on the faults and the location at which these faults are injected, this hardware-based fault injection can be classified to two sub-categories [3, 4, 10]:

- Hardware fault injection with contact. This injection has physical contact with the target system to directly effect on the system. This is the most common method of hardware-based fault injection. The idea behind this method is how to alter electrical currents and voltages at the pins. There are two main solution existing.

1) *Active probes*. This technique can add current via the probes attached to the pins, altering their electrical currents. This method can mainly emulate the stuck-at faults. However, the damage of target hardware system can easily happen if the electrical currents cannot be properly controlled.
2) *Socket insertion*. This technique inserts a socket between the target hardware and its circuit board. The inserted socket can inject stuck-at, open or more faults into the target hardware by forcing to alter the analog signals that represent the corresponding logic value onto the pins of the target hardware.

Both of these methods provide good controllability of fault times and locations with little or no perturbation.

- Hardware fault injection without contact. This injection has no directly physical contact with the target system. Instead, there will be the external source to generate such as radiation to affect the signal inside the system. The advantages of this technique are, there is less chance to damage target hardware system, and it can mimic natural physical phenomena. However, it is hard to identify the time and location of faults, since radiation needs to be trigged and is hard to control the exact moment of emission.

Some tools that implemented this hardware-based fault injection are:

- MESSALINE[38]. This tool uses both active probe and socket insertion to implement the pin-level fault injection. It can inject stuck-at, open, bridging and more complex logic faults. The highlight of this tool is to have a special device that can monitor whether the injected fault is active.
- FIST [43]. This tool takes both contact and contactless methods to create transient fault inside the target system. It equippes a device to generate a special radiation to create transient fault in random location inside the target system.
- MARS [45]. This tool is significantly similar with FIST, besides, it uses electromagnet to conduct cantactless fault injection.

3.1.3.4 Software-based Fault Injection

This type of injection is more attractive in recent years as it does not require expensive hardware for experiments. Another advantage to apply software-based fault injection is that, the particular fault can be inserted into the simulated hardware application, which may not be accessible in practice due to some safety reasons [23, 24].

If the target is software application, the fault can be injected into the application itself or layered between the application and the operation system. If the target is operation system, the only approach is to embed injector into the system, since it is difficult to add a layer between the injector and operation system.

Although the software approach is flexible, it also has some disadvantages [4].

1) It cannot inject the fault to the location, where it is inaccessible to software.
2) The software may disturb the workload of the system, which will cause the inaccuracy of the simulation.
3) The simulation time consuming will happen to some long latency faults.

Software-based fault injection also can be divided into two classifications [4, 9, 10, 11]:

- Software-implemented fault injection. This method has been implemented as sort of tools, by which a set of faults such as mutated instruction stream will be injected into the software of the hardware system and measure the ability of the system to detect, locate and recover from errors. Such relative tools are FIAT, FERRARI[16]. This method, however, may disturb the workload running on the target system or even change the structure of the original software [3]. Furthermore, some locations of hardware may not be accessible for the software so that the fault cannot be inserted into a desired location.
- Simulation-based fault injection. This method models the behaviour of the target circuit and imitates the simulation [27, 35, 39, 40, 49]. This type of fault injection, compared with software-implemented fault injection, essentially overcomes the defect and improves the controllability of when and where to inject faults. The behaviour of the circuit can be described by using hardware description language such as Verilog and VHDL, or even native language such as C/C++. Such

implementation is the MEFITSO [47]. A very critical issue for the simulation-based fault injection is how to precisely represent a physical circuit by a model. If the model is not correct or valid, the result derived from this model will not be valid either.

3.2 Fault Injection to Verilog Circuit Model

Some approaches recently are used to inject stuck-at fault into Verilog circuits. In [8, 10, 11, 46], two different automatic fault injectors are proposed. The fault injector described in [11] uses *force* and *release* statements of Verilog to apply logic value 1 or 0 to each line of tested circuit, and simulates this faulty circuit to obtain faulty outputs. This fault injector directly modifies and then compiles the circuit file in Verilog format in a commercial simulator. Its primary inputs can be fed by either pseudorandom test patterns or deterministic test patterns from external source such that the test pattern is generated by COMPACTEST [54]. The fault coverage reaches 100% if the deterministic test patterns are applied. However, this approach cannot simulate sequential circuits as it does not have any mechanism to deal with clock and flip-flops of sequential circuits.

The other fault injector proposed in [8, 10, 11] is capable of injecting faults into both combinational and sequential circuits. It generally consists of two parts: the hardware simulation environment Altera MAX PLUS II and software based fault injector. The hardware simulation environment simulates the circuit under test and enables it to perform both fault-free and faulty testing. On the other hand, software based fault injector firstly generates the test patterns for circuit under test, and then physically replaces the target wire (i.e. fault location) by 0 and 1 (representing stuck-at-0 and stuck-at-1) inside the Verilog file of the circuit. Compiling this Verilog file every time is very necessary after a fault is injected. The faulty circuit is simulated by Altera MAX PLUS II and then output is compared with fault-free output to determine whether the fault is detected. This approach can achieve reasonable fault coverage for both combinational and sequential circuits. The simulation time, however, is very long due to the compiling faulty Verilog file.

Chapter 4

Fault Injection Under Software and Hardware Co-design Environment

4.1 Hardware Design Environment

4.1.1 Introduction to Verilog HDL

A complete testing environment is a synthesis of hardware and software. The test hardware generally contains a set of digital circuit, power supplies and measurement device. In this thesis, all the hardware used in our experiment is represented by Verilog HDL. Hardware description language (HDL) is a specific computer program language that has been used to describe the circuit's operation, design and organization in the text-based format. It is the most important modern tools used to describe hardware. The principal feature of the HDL is that, it contains the capability to describe the function of a piece of hardware independently of the implementation, as well as allows the entire design to be taken place in a single program language. Within HDL, hardware designer directly call different logical gates that have been defined previously to form the desired function blocks. Moreover, HDL can model multiple processes such as flip-flop and multiplier independently of one another. The compiler is eventually called to synthesize the entire functional blocks to implement a functional circuit.

There are two most widely used and well-supported HDL in recent industries: VHDL and Verilog HDL [56]. They are essentially designed to build efficient simulation of digital systems. This intention causes some fundamental differences from other hardware description languages such as EDIF that is designed to describe structure of nets and components used to build a system. Furthermore, both VHDL and Verilog HDL are used to define the functionality of components or high level abstraction for logical synthesis. This synthesized model can also be simulated in the simulator to check whether the functions designed are running properly before it goes to physical synthesis. The most common mode of synthesis is register-transfer level synthesis, which uses logic synthesis on the

combinational logical blocks to optimize their implementations. Moreover, VHDL and Verilog HDL both are built on the same basic frame work of event-driven simulation. Event-driven simulation is an efficient method for hardware simulation, because it takes advantage of activities of the circuit as not all values on each component are changed on every clock cycle and ignoring inactive components can reduce the simulation time.

4.1.2 VHDL

VHDL has been developed as a general-purpose programming language as well as a design-entry language for field-programmable gate arrays (FPGA) of digital circuits. It became IEEE standard 1076 in 1987 and updated in 1993 know as IEEE standard 1076-1993. It allows design engineers to describe and verify the behaviour of required system before synthesis tools translate the design into real hardware. Recently lots of free FPGA synthesis tools are available in the market with VHDL simulators, although some of them are limited in some functionality.

4.1.3 Verilog HDL

Verilog HDL, usually called Verilog, was firstly invented in 1985. By now, the latest version is Verilog 2005 defined in IEEE Standard 1364-2005. The Verilog enables hardware designers to describe a complex digital circuit as a module within a wide range of levels of abstraction, either by top-down or bottom-up design [9, 41]. The module specifies a closed system that contains both test data and hardware models. Furthermore, Verilog supports mixed-level design, allowing behavioural design and structural design coexisting. A behavioural specification refers to the behaviour of a module using program language constructs, while the structural specification describes a module containing a hierarchical interconnection of submodules. A simulation is applied to verify the design of digital circuits in Verilog. To simulate a digital circuit in Verilog, a testbench is written, which includes an instantiation of the digital circuit. Some test data as given inputs to the instance of digital circuit are applied in the testbench after the corresponding parameters were setup, and then the reaction of the digital circuit is observed and compared with the reference. The code of C17 benchmark circuit in the form of Verilog HDL is shown as follow.

```
// Verilog
// c17
// Ninputs 5
// Noutputs 2
// NtotalGates 6
// NAND2 6
module c17 (N1,N2,N3,N6,N7,N22,N23);
input N1,N2,N3,N6,N7;
output N22,N23;
wire N10,N11,N16,N19;
nand NAND2_1 (N10, N1, N3);
nand NAND2_2 (N11, N3, N6);
nand NAND2_3 (N16, N2, N11);
nand NAND2_4 (N19, N11, N7);
nand NAND2_5 (N22, N10, N16);
nand NAND2_6 (N23, N16, N19);
endmodule
```

The Verilog HDL code above can be translated by the synthesis tool to a graph of netlist of the circuit, shown in Fig. 4.1.

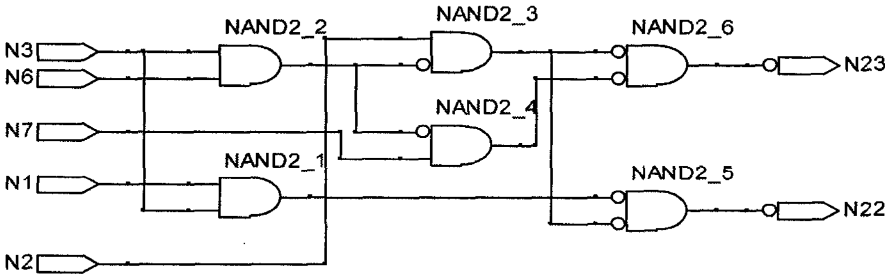

Figure 4.1 Netlist of C17

4.1.4 Testbench

Any digital circuits have to go through a testing process before going to the end-user. The reasons that make testing involved are [7, 13, 25]:

1) Check and ensure all functions designed are carried out as desired. A set of values are applied at inputs and the values at outputs are observed.

2) Check and ensure all functional sequences designed are carried out as desired. A set of time-based sequential values are applied at inputs and the values at outputs are observed.

3) Check the timing behaviour. A set of various values are applied and the time delay and/or pulse width are observed.

The idea behind the testbench approach is based on the requirement for which the system being designed wires to a test generator that will provide inputs in controlled time frame and display outputs. Fig. 4.2 below gives a general view of testbench module.

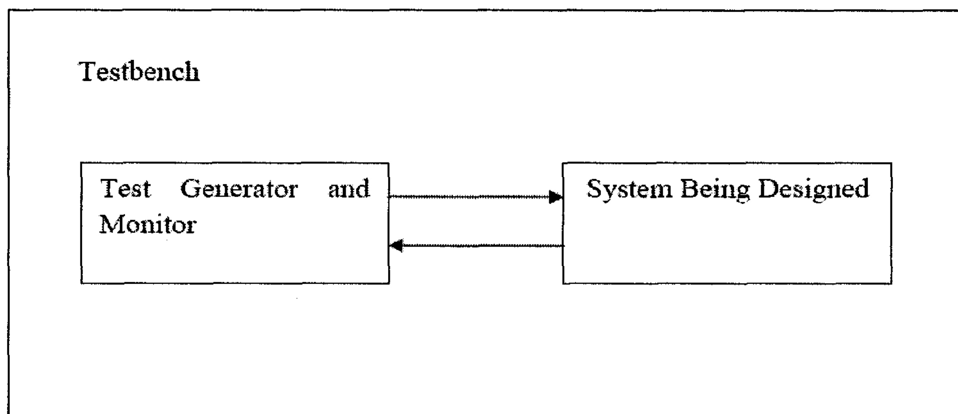

Figure 4.2 General view of testbench module

4.1.5 ModelSim with PLI Design Environment

ModelSim is a popular commercial verification and simulation tool for VHDL, Verilog and other HDL languages [6]. ModelSim offers text editor, design optimization, simulator, waveform analysis, signal tracing, and PLI/VPI interface. Several different design

applications can be activated simultaneously. Meanwhile, other applications can be run on the background. Among these applications, we will focus on the simulator and PLI/VPI applications. Programming Language Interface (PLI), which was firstly introduced in 1985 as part of a digital simulator called Verilog-XL developed by Gateway Design Automation. PLI then were mostly deployed in Gateway's simulator products. Due to the powerful capabilities of PLI, Verilog-XL became one of the most popular digital simulators used by hardware engineers. During this period, Gateway recognized the potential of PLI, as well as the industries. Recently, PLI is defined in IEEE 1364-2001, which proposed a number of substantial enhancement based on the previous version IEEE 1364-1999.

PLI indeed is a mechanism to provide an Application Program Interface (API) to Verilog. Essentially it is a mechanism to invoke C/C++ functions from Verilog code [6]. PLI enable engineers to create their own programs to access the internal structure of commercial simulator. Through this interface, PLI is provided as a mechanism for applications to access the description and to control the behaviour of the tool. Hence, a commercial simulator can be customized to perform any tasks desired. The highlights of PLI include [13, 25, 32]:

- Functional models transferred. Some hardware models are represented in C language instead of Verilog HDL. PLI provides many ways to transfer these models back to Verilog in order to use Verilog simulator.
- Access to programming language libraries. PLI allows Verilog model to access the libraries of C language such as math function and have the returned arguments.
- Reading test vector files. The I/O function in C language can be easily deployed through PLI in Verilog model to test vector files for Verilog simulation.
- Delay calculation. Through PLI, the delay calculator can be designed in C language and used in Verilog model to calculate various outputs of this model after simulated.
- Customize output displays. By using PLI, Verilog HDL can provide outputs in different formats that are tailored for the end-user.

- Design debug utilities. A C program using PLI can be written to collect necessary information of Verilog model to debug a Verilog design

- Simulation analysis. Most Verilog simulator just simply applies test patterns to a model and displays resulting model outputs. The PLI function can significantly enhance this simulation by analyzing details on the simulation inside such as generating different technical reports on demand.

The first generation of the PLI was the "task-function routines" also called as "TF interface" developed in 1985 [55]. These routines were mainly used to retrieve arguments of user-defined system tasks or functions in Verilog simulator. This TF interface primarily consists of three callback functions:

- Checktf – callback during compilation

- Calltf – callback during the execution of system task or function

- Misctf – callback for a specific reason defined by user

The second generation of PLI was the "access routine" also called as "ACC interface". The major improvement of ACC interface is the ability to modify and propagate values of regs, as well as callback for any value changed.

The function invoked in Verilog code is called *system call*. Also PLI allows users to define their own system calls, which are neither pre-defined nor allowed in Verilog syntax such as custom output display, testbench modeling and simulation analysis. The reason to use PLI is because 1) pure Verilog is unable to express desired behaviour; 2) pure Verilog is unable to model deep submicron; and 3) pure Verilog is unable to model high level function [6]. A typical PLI working procedure is shown in Fig.4.3.

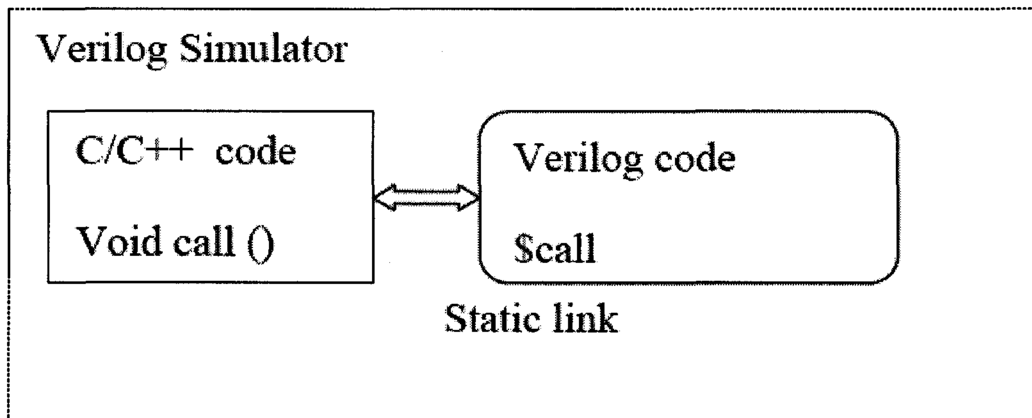

Figure 4.3 Typical PLI working procedure

After the user defines the function in C/C++, this function will be compiled and a shared library .DLL will be generated. The ModelSim can recognize the user-defined system call and link to corresponding .DLL file, integrating the C/C++ function with simulator. During the execution of the Verilog code, whenever the simulator encounters user-defined system call, the execution control will be passed to C/C++ function.

4.2 Hardware and Software Co-design Framework

The interest in hardware and software Co-design has been raised due to the introduction of computer-aided design (CAD). Co-design is termed as a joint development of hardware and software component in order to obtain a complete hardware architecture and software executing on the hardware. One of the goals of co-design is to shorten the time-to-market while reducing the design effort and costs of the products. To achieve hardware design, the process always follows the top-down design methodology, shown in Fig. 4.4. This design flow allows design engineers to implement an actual hardware step by step based on the design requirements. We generally represent all designs in Verilog at three levels, behavioral level, functional level and gate level. In behavioral level, the algorithm of digital system is constructed without considering actual logical gates. Functional level presents the hardware registers and combinational logic. At this level, the description can be translated into hardware by synthesis tools. Gate level eventually implements explicit description of logic primitives and interconnects among them.

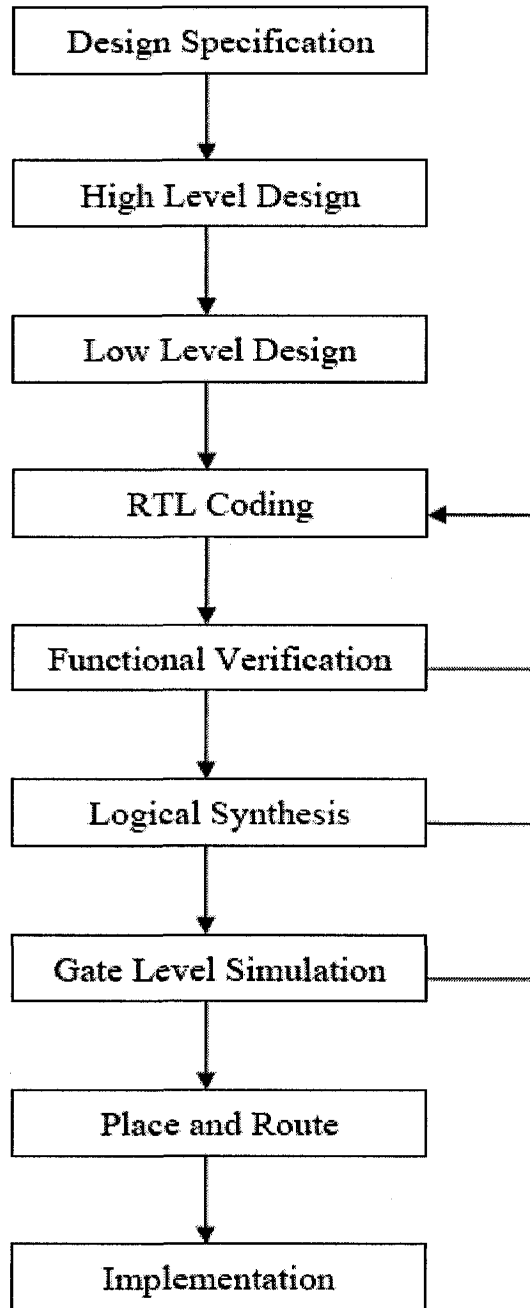

Figure 4.4 top-down design flow

In this thesis, test software is able to provide the automation of entire testing process. The test program embedded is used to control the test function, generate test patters, apply test patterns to given MUT, collect the response output and compare the output with fault-free

output stored on a chip. General speaking, the test software will complete the whole test simulation automatically. The design process in this thesis is based on a cores-based embedded approach [8] and deals with hardware and software co-design. The goal we will achieve is to implement combinational and sequential logical circuit testing and simulation environments and verify this design scheme in order to prove it is correct. The method we use to verify our design is the design for testability approach, which is one of the most efficient and reliable approaches. Our aim at this design is to capture as many as possible injected faults at the output of MUT. We further will extend this design to different operation environments. This program is designed to deal with both ISCAS85 and ISCAS89 benchmark circuits, which are illustrated by gate level Verilog HDL. In this approach, PLI is chosen due to the fact that it can directly access the data structure of ModelSim simulator to control the entire testing function. Fig 4.5 indicates the brief design of PLI test function. This test program mainly consists of four functional blocks:

1) Test patterns generation,

2) Faultlist generation,

3) Fault injection, and

4) Comparator unit.

Meanwhile, in ModelSim simulation environment, the benchmark circuit is given in the form of Verilog HDL, while the testbench is implemented for corresponding benchmark circuits.

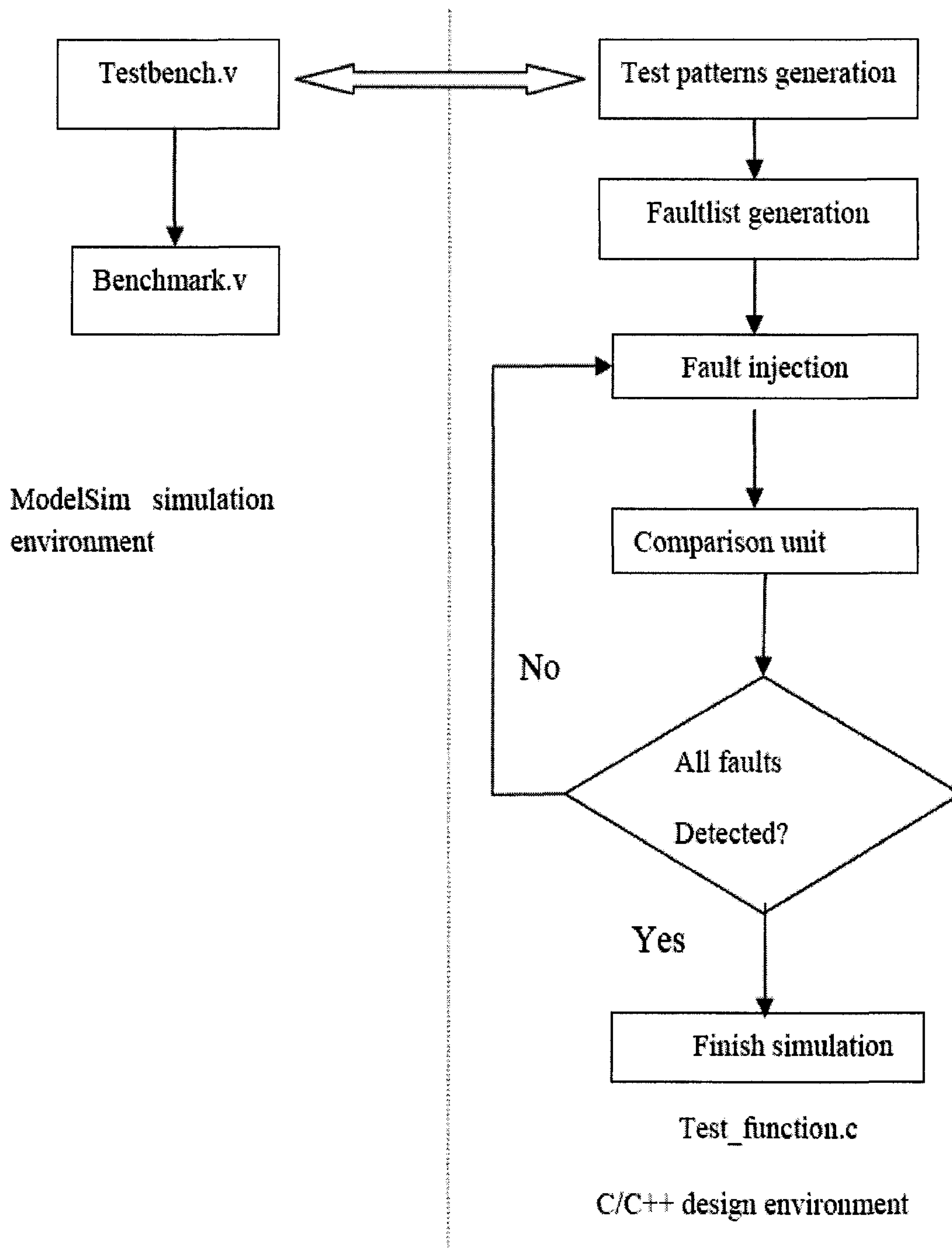

Figure 4.5 Design flow of fault simulator

Chapter 5

Design and Implementation of PLI Fault Injection for Combinational Circuits

5.1 General Structure of Combinational Circuits

A combinational circuit is the one whose outputs are determined solely by the values of the inputs. It appears to combine with the inputs in some way to produce the outputs. It can be represented by a truth table and compute a Boolean function. Such circuit has three main characteristics [15]. Firstly, this circuit consists of a minimum number of gates. Secondly it may contain complex gates. And thirdly, it has a minimum number of levels of gates. The first two characteristics are presented by minimizing the number of gates inputs, which further has the effect of minimizing cost. Minimizing the number of levels of gates can reduce the circuit delay and make the circuit faster. Hence, for simple function, it is possible to design a circuit by inspecting its function. For more complex functions, we have to look into the details on each Boolean function to compose the whole circuit function.

Some combinational circuits are playing very important role in the construction of the processor and memory unit. Such combinational circuits include Multiple-Input AND, Comparator, and Decoder.

The hardware for the design environment includes a personal computer with AM2 3000+ and 3.5 GB of RAM. The involved software for the design environment contains program language C, Verilog HDL that embeds PLI, and a commercial simulator ModelSim 6.3. In our implementation, by applying PLI, our fault injection approach implemented in C can access the CUT (Circuit Under Test) described in Verilog HDL and simulated in ModelSim. Each wire in CUT will be stuck at either 1 or 0 using Stuck-At fault model. The primary output exported from ModelSim will be caught and analyzed. A fault is reported as long as its value is different from the reference value

5.2 Design of PLI Fault Injection

5.2.1 Overview of Design of PLI Fault Injection

Fig. 5.1 depicts the design flow of this fault injection approach for combinational circuits.

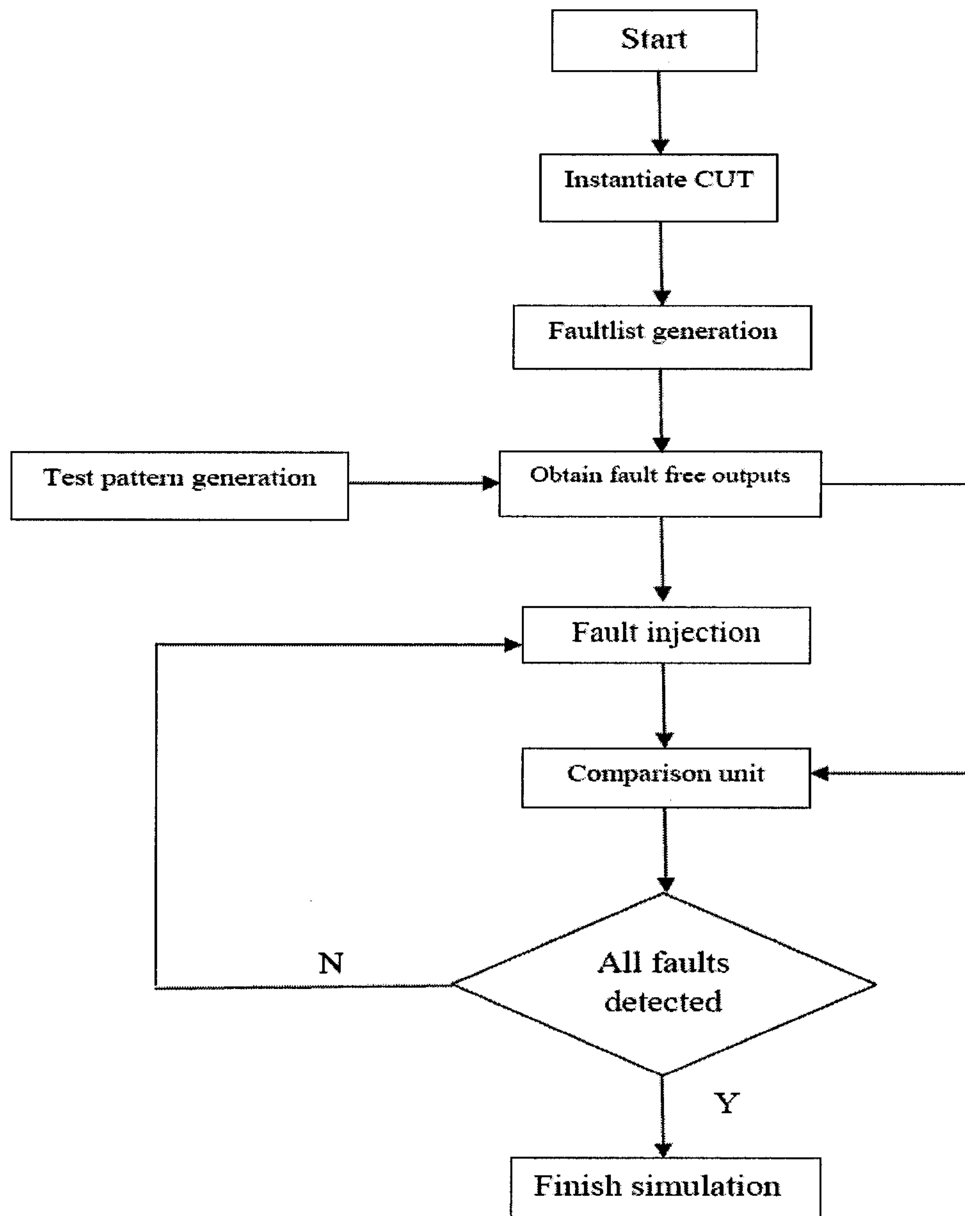

Figure 5.1 Design flow of PLI fault injection for combinational circuits

The entire design of fault injection for combinational circuit generally consists of four functional blocks: 1) circuit under test (CUT); 2) pseudorandom test pattern generator; 3) fault injector; and 4) comparison unit. These four functional blocks are essential elements for most fault simulator.

In this design environment, we mainly focus on the CUT, ISCAS 85 benchmark circuit, implemented in the form of Verilog HDL [20]. An example of C17 is shown as follow.

```
// Verilog
// c17
// Ninputs 5
// Noutputs 2
// NtotalGates 6
// NAND2 6
module c17 (N1,N2,N3,N6,N7,N22,N23);
input N1,N2,N3,N6,N7;
output N22,N23;
wire N10,N11,N16,N19;
nand NAND2_1 (N10, N1, N3);
nand NAND2_2 (N11, N3, N6);
nand NAND2_3 (N16, N2, N11);
nand NAND2_4 (N19, N11, N7);
nand NAND2_5 (N22, N10, N16);
nand NAND2_6 (N23, N16, N19);
endmodule
```

As we previously mentioned, each module has to associate with a testbench for verification. In our approach, we do not need to verify the module. But instead, we use the testbench as a bridge to send the test data to CUT. The testbench is linked to Test_function.c via PLI interface, and firstly used to apply the test patterns generated from test patterns generation of Test_function.c. Due to the different length of input stream of each benchmark circuit, the testbench has to be customized in order to give appropriate length of input.

Faultlist generation block of Test_function.c is applied to generate a complete fault list of given benchmark circuit. In this thesis, we only consider single stuck line (either stuck-at 0 or stuck-at 1) fault model for the gate-level circuit. Single stuck line fault model is widely used for most manufacturing testing. This fault model assumes that each time one line in the digital circuit is stuck at logic value 1 or 0. When a line is stuck, it is called a fault. The objective of running this Faultlist generation function is to firstly identify the fault location, where a stuck-at fault will be injected, and secondly provide the total number of possible stuck-at faults that will be used for the calculation of fault coverage. Fault injection is a function that is in charge of inserting stuck-at faults. This function will be activated after the corresponding fault-free output is captured and stored from the same test pattern. The comparator function is designed to distinguish the faulty output whether it is different from the fault-free output previously captured. Only the difference detected after comparison will be reported as a detected fault. This detected fault will later be removed from the fault list in order to narrow the undetected faults on the fault list for next fault injection. Control signal will be given from Test_function.c to determinate the number of test patterns to be applied into MUT. The condition to terminate the test simulation is either all the faults are detected or the number of test patterns applied reaches the value of control signal. Fig. 5.2 describes the detailed structure of PLI fault injection approach running in our design environment.

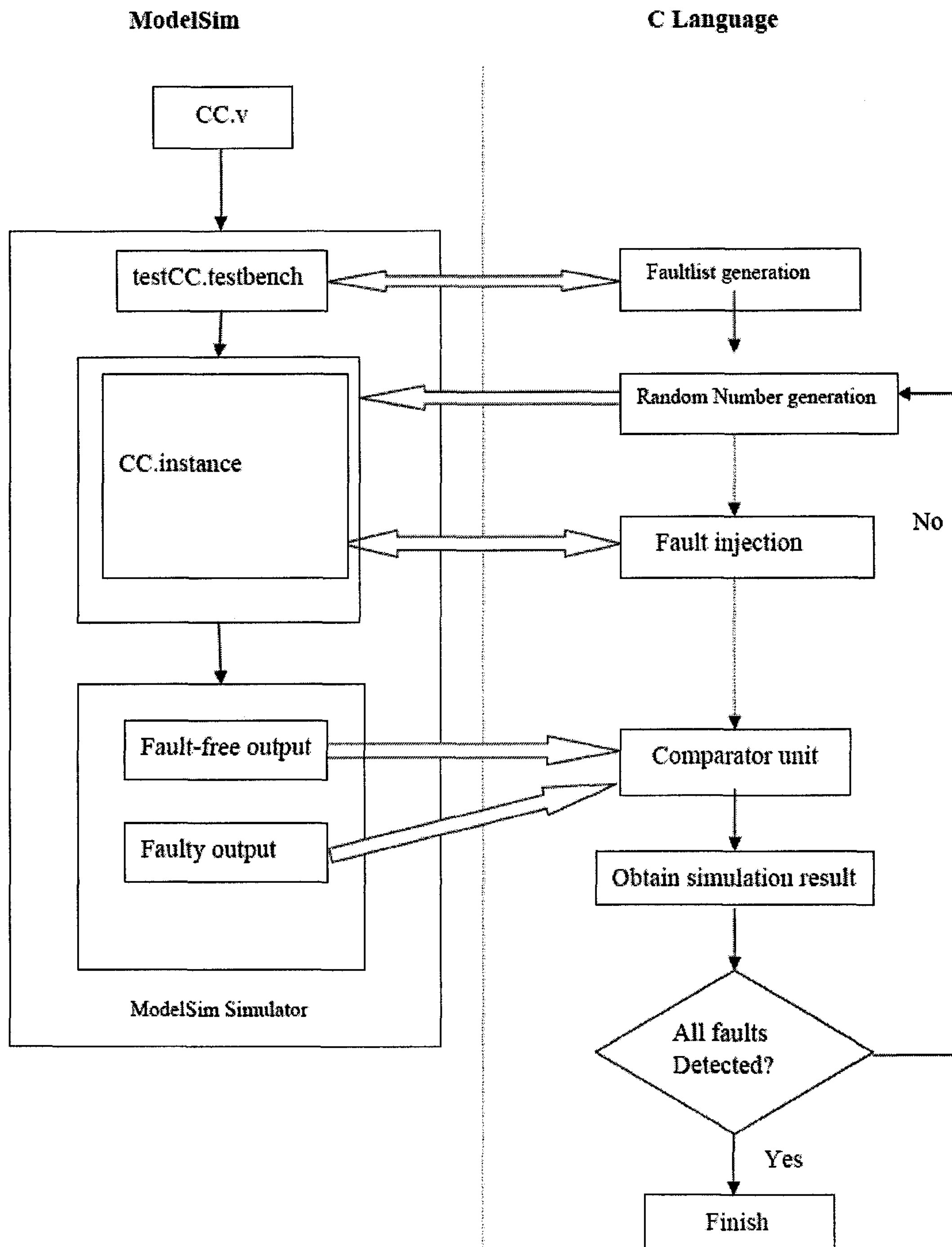

Figure 5.2 Structure of PLI fault injection

5.2.2 Initialization of Fault Injection

The task of initialization is to reset the circuit and the fault injector. After initialization, we can have both target circuit and fault injector prepared.

5.2.3 Random Test Pattern Generation

In this paper we prefer to choose pseudorandom test pattern generation method. The selection of test pattern generation method has significant impact on the performance of the whole testing, since this method has less computing overhead than deterministic test pattern generation method such as D-algorithm [2], which has to access the internal structure of digital circuit to determinate the specific value in primary input.

In BIST filed, linear feedback shift register (LFSR) is widely used as a hardware random signal generator. A LFSR is driven by clock and can be formed by performing exclusive-OR on the outputs of two or more of the flip-flop together and feeding those outputs back into the input of one of the flip-flops. It is an ideal pseudorandom test pattern generator, because when the outputs of the flip-flops are fed by seed value, and when the LFSR is clocked, it will generate a pseudorandom pattern of 1s and 0s. Fig. 5.3 shows a typical LFSR.

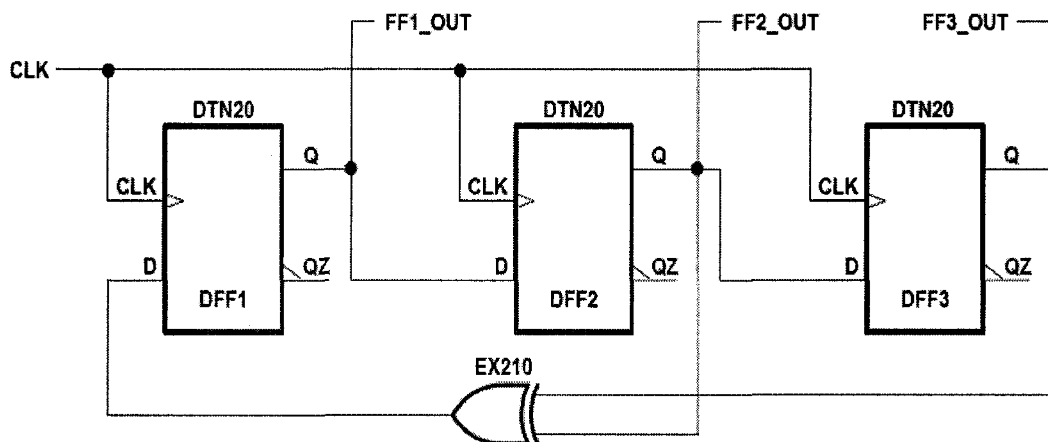

Figure 5.3 Linear Feedback Shift Register

In this thesis, we focus more on the software implementation of pseudorandom test pattern generation. The pseudorandom test pattern generator module is to generate a

sequence of pseudorandom numbers that can feed the input of given circuit. Normally, the maximum decimal number that can be fed into the input of benchmark circuit is 2^m-1, where m is the number of the primary inputs of given circuit. The algorithm of the pseudorandom test pattern generator is shown as follow:

```
Static void test_pattern_generator
{
Identify the length of given circuit;
Calculate maximum decimal number d;
Catch current time as random generation seed;
Random number =rand()%d
Send random number to testbench.v of ModelSim;
}
```

This algorithm generates pseudorandom decimal number between 0 and d. The pseudorandom number sent to ModelSim will be automatically converted to binary format in ModelSim to feed the input of given circuit. We also can input external deterministic test patterns in the form of text file as the input of given circuit. The pseudorandom test pattern for C17 circuit is shown as follow:

```
# Input vector ---  01001   Fault free output:   11
# Input vector ---  00011   Fault free output:   01
# Input vector ---  11110   Fault free output:   10
# Input vector ---  00100   Fault free output:   00
# Input vector ---  00001   Fault free output:   01
```

In this output file, the input vector, 01001, for example, corresponds to the primary inputs N1, N2, N3, N6, N7 of the circuit, while the free output 11 means the primary outputs N22, N23.

5.2.4 Fault List Generation

In this approach, we simulate gate level circuits using stuck-at fault model. Based on this fault model, we assume that each line inside the circuit has two faults, SA0 and SA1. Fault list is generated to identify the location where a stuck-at fault will be injected. The logic value 1 and 0 will be set in each line of a given circuit, respectively. Hence, the fault list collects all the lines of the circuit that will have the fault injection. Also there is a counter accompanying the fault list generation, which indicates the total number of faults on this list. The fault list generated for C17 circuit is shown as follow:

```
# Generating fault list........
# Fault 0  list ----  N1
# Fault 1  list ----  N2
# Fault 2  list ----  N3
# Fault 3  list ----  N6
# Fault 4  list ----  N7
# Fault 5  list ----  N22
# Fault 6  list ----  N23
# Fault 7  list ----  N10
# Fault 8  list ----  N11
# Fault 9  list ----  N16
# Fault 10  list ----  N19
# ----------Fault list has been generated-------------
```

5.2.5 Fault Injection

Fault injection is the most important part of the entire simulation. The hardware fault injection technique is essential in order to iteratively injection stuck-at faults to every single

wire. A manner of hardware fault injection is displayed in Fig. 5.4 [19]. In this scheme, several multiplexers are used to inject a stuck-at fault to the target wire according to the operation signal.

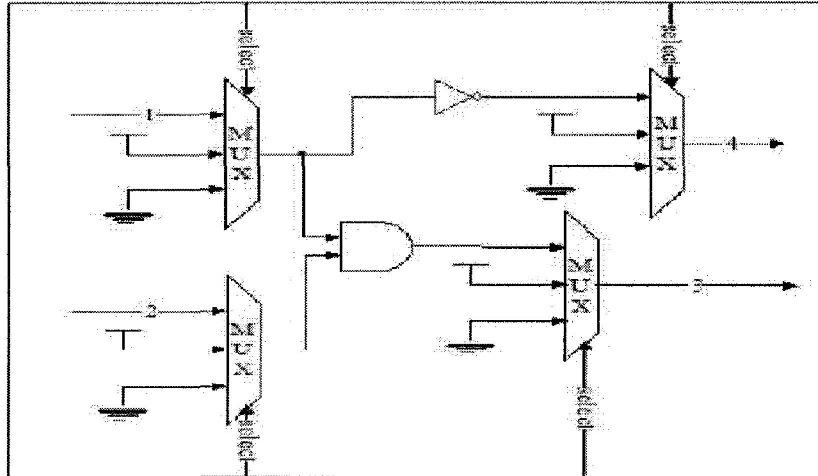

Figure 5.4 hardware fault injection scheme [19]

Software fault injection, on the other hand, is to set each line with logical value 1 and 0, sequentially and respectively. To take advantage of PLI with ModelSim simulator, we can treat the circuit as a white box. For primary inputs, primary outputs and internal wires, we can use the same fault injection method to change the real value without modifying the structure of a specific module of the circuit during the injection.

In order to realize the fault-free response from MUT, before fault injection function is applied, a fault-free output is captured and stored in comparator unit for the later comparison. Then the fault injection function will locate the line given in fault list and inject the fault. The algorithm for fault injection is shown as follows:

```
Static void fault_injection()
{
Capture fault-free output for comparator;
if(fault-free output is captured)
{
Select a line to inject fault;
Switch(inject_fault_type)
{
Case: stuck-at-1
accForceFlag logical value 1 at selected line;
Case: stuck-at-0
accForceFlag logical value 0 at selected line;
}
Capture faulty output;
accReleaseFlag stuck-at value at selected line;
}
}
```

In the algorithm above, *accForceFlag* and *accReleaseFlag* are two APIs provided in PLI. The *accForceFlag* is defined to force a value into a net, overriding any existing values. The value written into a net has the same manner as the *force* statement in Verilog. Only one forced value can be set at a time, regardless of whether the forced value was set previously by PLI or Verilog. The *accReleaseFlag*, on the other hand, is used to release any forced value existing in the line. It has the same function as the *release* statement in Verilog as well.

The stuck-at fault will be injected into a wire sequentially, and then released immediately after the faulty output is captured. One test cycle, which is defined as a complete procedure of applying one test pattern to detect all undetected faults of a given circuit, is finished after

detected faults are removed from fault list. Another test cycle is started with a reduced fault list, which will highly improve the performance of simulation.

5.2.6 Comparator Unit

In the forepart, the pseudocode description of fault injection algorithm that can insert stuck-at faults into inputs, outputs and internal wires was given. The faulty output is captured and then compared with fault-free output by comparator unit to identify whether the injected fault is detected. If the fault is not detected, the program will take next test pattern to run the simulation until all the faults are detected. All the results are saved in the file "transcript.file" generated by ModelSim. The detailed pseudocode algorithm for comparator unit is given below.

```
Static void Comparator ()
{
Receive faulty output;
Switch (injection type)
{Case SA0: if mismatched, report this fault and remove it from fault list;
  Case SA1: if mismatched, report this fault and remove it from fault list;
}
If (not all faults detected)
{require to generate a new test pattern}
}
```

5.2.7 Testbench Design

After setting up the PLI program in C/C++ environment, we turn out focus on the testbench design in ModelSim simulator. Inside the testbench, the ISCAS85 benchmark circuit by Verilog is instantiated. The test data from PLI is applied to the benchmark circuits via this testbench. The final test results also are monitored and saved through this testbench. The sample of testbench for C17 is shown as follow:

```
module testc17();
   wire[1:0]o; //output ports
   reg[4:0]i;  // input ports
c17 c(i[4],i[3],i[2],i[1],i[0],o[1],o[0]); //instantiate C17 circuits
  initial
 begin
 $Get_faultlist(c);
end
 always
 begin
 $Fault_injection(c);
#1 i=$randomvector(c);
     $Get_output(o); // capture the simulation results
     end
   initial
  #($control) $stop; //receive control signal to control test cycles
Endmodule
```

In this testbench file, *$Get_faultlist(c)* responses to fault list generation of PLI, giving a complete nets of the circuit. *$Fault_injection(c)* corresponds to fault list generation of PLI to insert the stuck-at fault to each net. After the forced value was set, *$randomvector(c)* receives the test pattern from test pattern generator of PLI. *$Fault_injection* is always run before the test pattern is applied in MUT in order to avoid the propagation delay in the circuit. *$Get_output(o)* eventually captures the simulation results that are sent back to comparator unit of PLI to compare the outputs.

Chapter 6

Design and Implementation of PLI Fault Injection for Sequential Circuits

6.1 General Structure of Sequential Circuits

In previous chapter, we said that the primary output of combinational circuits depends solely upon the input. The implication is that combinational circuits do not have memory embedded. For the purpose of building sophisticated digital circuits for various requirements of devices, we need a specific type of circuit whose output can depend on both the input of the circuit and its previous state, which means memory.

As we previously mentioned, there are different types of sequential circuits such as S-R Latch, Clocked D-Latch, Master-Slave Flip-Flop, and Edge-Triggered Flip-Flop. In this thesis, we just focus on the sequential circuits with Master-Slave D Flip-Flop. The full-scan ISCAS 89 sequential circuits are used as benchmark circuits [20].

The hardware for the design environment includes a personal computer with AM2 3000+ and 3.5 GB of RAM. The involved software for the design environment contains program language C, Verilog HDL that embeds PLI, and a commercial simulator ModelSim 6.3. In our implementation, by applying PLI, our fault injection approach implemented in C can access the CUT (Circuit Under Test) described in Verilog HDL and simulated in ModelSim. Each wire in CUT will be stuck at either 1 or 0 using Stuck-At fault model. The primary output exported from ModelSim will be caught and analyzed. A fault is reported as long as its value is different from the reference value.

6.2 Design of PLI Fault Injection

6.2.1 Overview of Design of PLI Fault Injection

Fig. 6.1 depicts the design flow of this fault injection approach. Due to the nature of sequential circuits, it is impossible to have a fixed fault-free output as the reference. The output is changed depending on current input and previous state. The number of states for

each sequential circuit varies. Hence, using state table to store each state's fault-free output is possible but it is time consuming and costly.

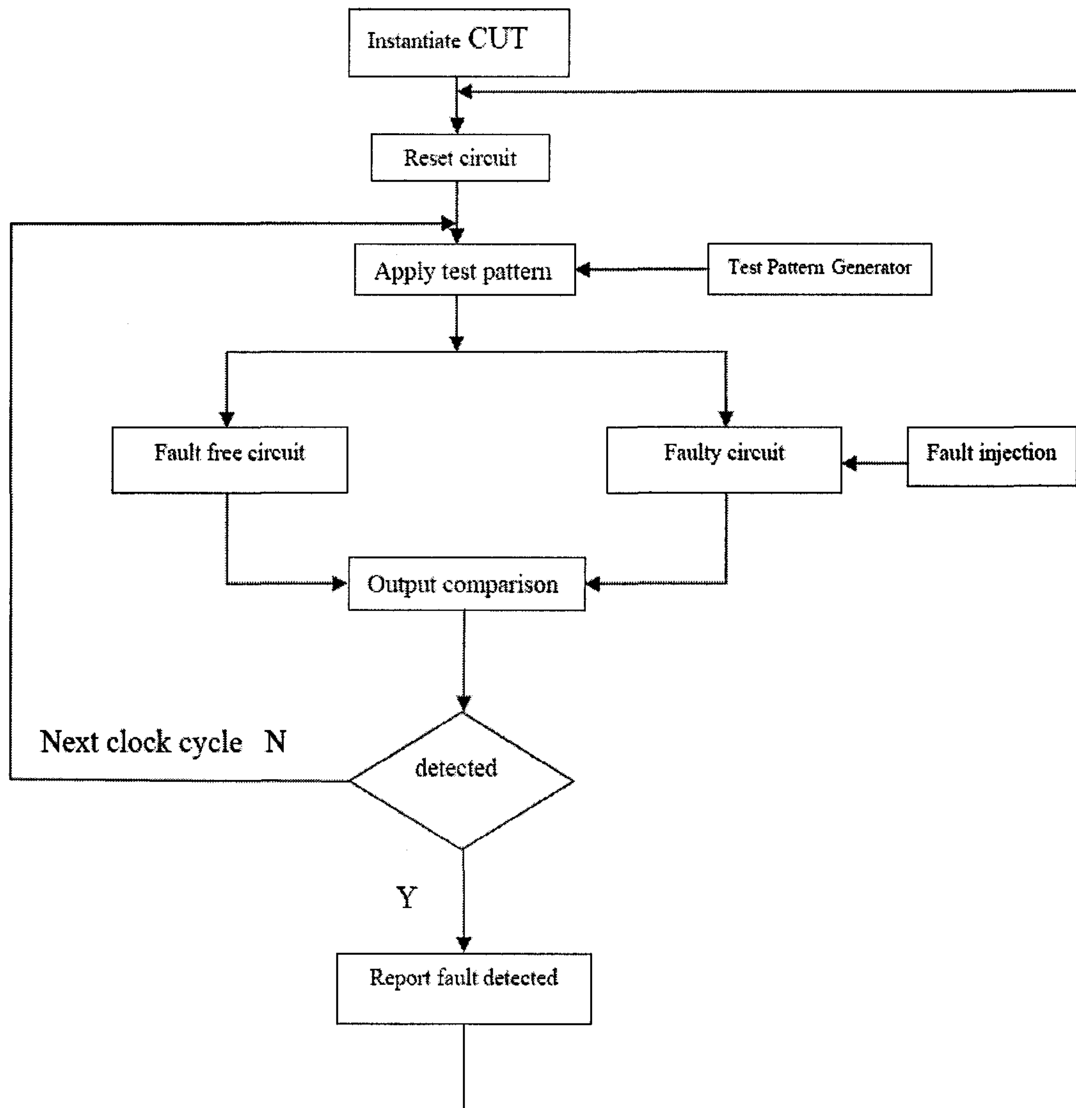

Figure 6.1 Design flow of PLI fault injection for sequential circuits

Therefore, the first issue that needs to be solved is how to reserve the fault-free output. PLI gives us a chance to simply solve this issue. In our fault injection approach, we provide an additional instance of fault-free circuit as a reference, which will run in the same clock cycle as the instance of faulty circuit does. This instance of fault-free circuit has exactly the

same structure as the one of faulty circuit. Each test pattern is applied to both instances of circuit, simultaneously. The only difference between these two instances is that the instance of fault-free circuit does not have fault injection. This difference further distinguishes the fault-free output and faulty output from each circuit. This means, firstly, if we do not inject any fault to either of these circuits, they must have identical states and primary outputs, and secondly, if a fault is inserted into the instance of faulty circuit, and this instance has different primary outputs, the fault is detected, since the primary outputs from instance of fault-free circuit are always considered as the fault free outputs. Fig. 6.2 describes the detailed structure of PLI fault injection approach running in our design environment.

6.2.2 Initialization of Fault Injection

The task of initialization is to reset the circuit and the fault injector, especially the clock that is reset to 0. After initialization, we can have both target circuit and fault injector prepared.

6.2.3 Random Number Generation

In BIST techniques, exhaustive, pseudoexhaustive, pseudorandom, or reduced test patterns are used because of the ease of their generation and storage on-chip [11]. On the other hand, some practical algorithms such as PODEM, FAN are used to generate reduced test patterns, which can obtain high fault coverage with low hardware cost.

In previous chapter, we mentioned that full-scan sequential circuit can be decomposed into flip-flops and a combinational logic block (CLB). The removal of flip-flops turns a full-scan sequential circuit into a combinational circuit. Therefore, the random test pattern generation for CLB is the same as the one for combinational circuits described in Chapter 5.

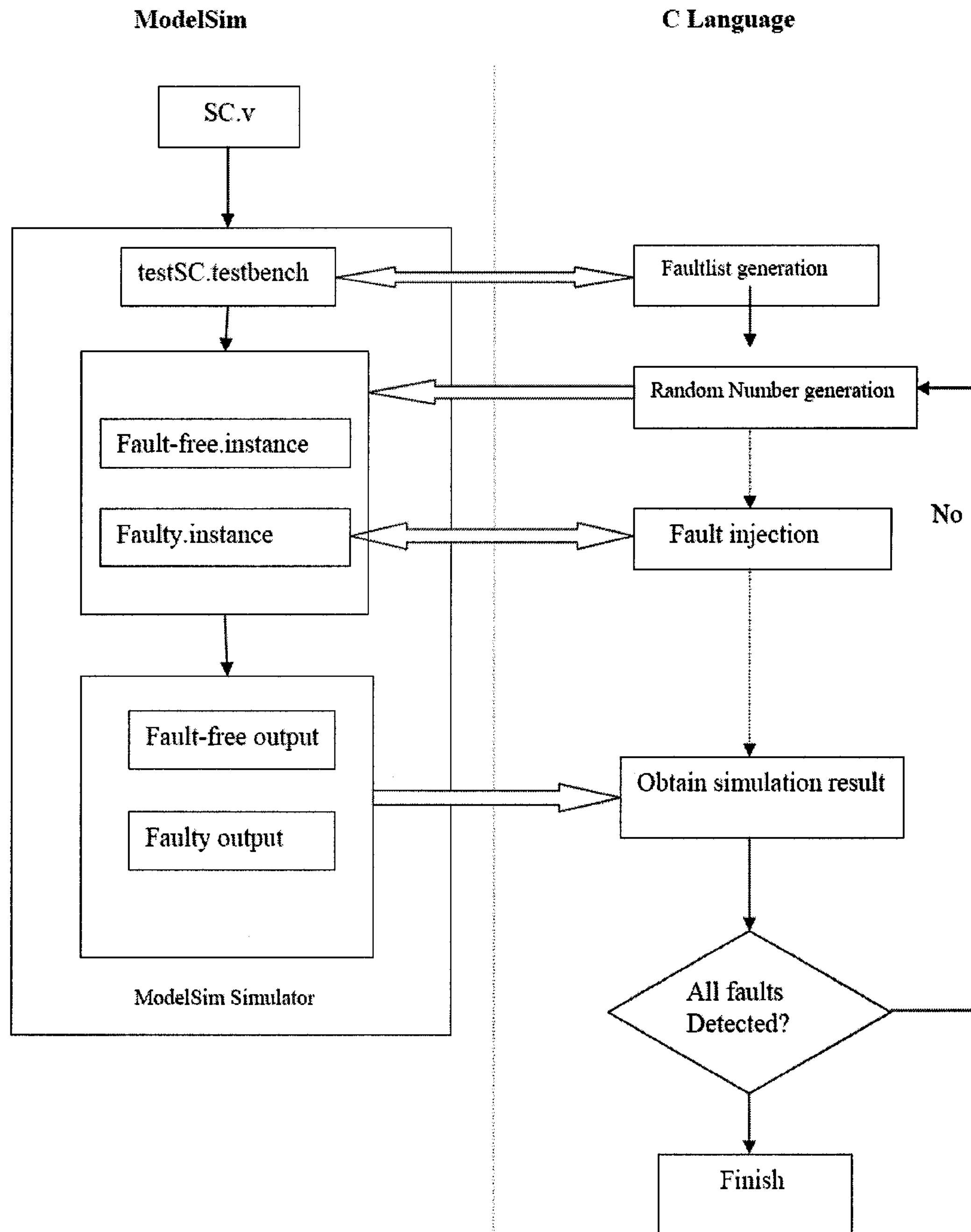

Figure 6.2 Structure of PLI fault injection

However, we still have to focus on implementing random number for the pseudoprimary inputs (PPIs) (i.e. the outputs of flip-flops). During the initialization state, flip-flops are

reset. If we do not assign the initial values on the PPIs, the circuit may take a couple of clock cycles to give PPIs initial values. The time taken on this process is depending on the scalability of the circuit. The larger the circuit is, the longer the time will take. This drawback may make the fault injection for sequential circuits more complex at the beginning. Our design of assigning initial values on the PPIs can overcome this drawback.

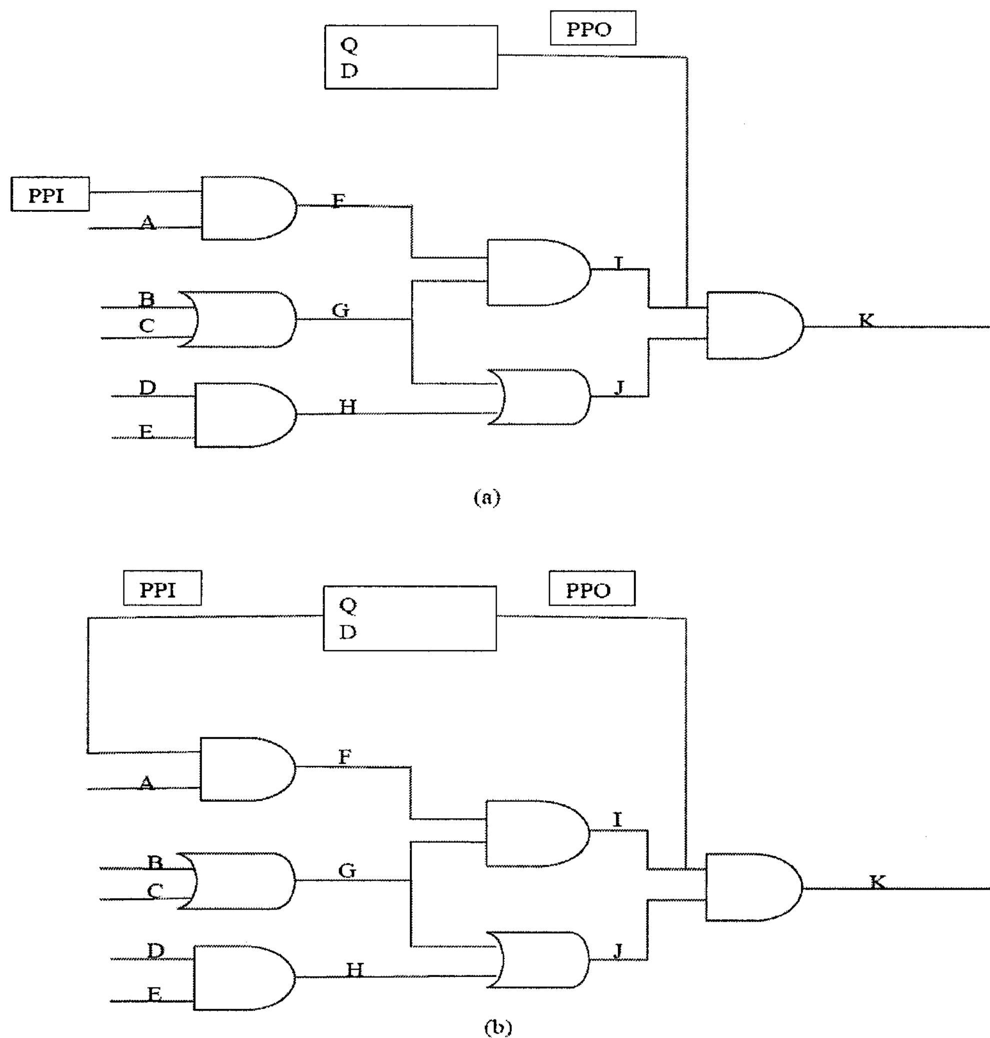

Figure 6.3 an example circuit. (a) Initializing PPI. (b) After initialization of PPI

Fig. 6.3 shows an example circuit, where we apply our approach to assign the initial value on PPI. In Fig. 6.3 (a), we are initializing the value on PPI. During this period of clock

0→1, the flip-flop is ignored. In this period, PPI is treated as one of primary inputs. The entire sequential circuit is considered as a combinational circuit, since the flip-flop does not affect the circuit behavior. The function *is_state_input ()*, listed below, is designed to select the PPI(s) of the target circuit.

```
static int is_statu_input(handle net_h)
{
loader=acc_next_load(net_h,loader);
driver=acc_next_driver(net_h,driver);

cell_l=acc_handle_parent(loader);
cell_d=acc_handle_parent(driver);

if(acc_fetch_name(cell_d)!=NULL&&strcmp(acc_fetch_name(cell_d),"P5")==0)
{
 value.format=accIntVal;
 delay.model=accForceFlag;
 value.value.integer=reset_value[reset_counter];
 acc_set_value(net_h,&value,&delay);
 reset_counter++;

return 1;

}

else

return 0;

}
```

This function automatically recognizes the PPI wires and assigns the generated value on each wire by applying *accForceFlag* API provided in PLI. This API is to force a value into a net, overriding any existing values. The value that is assigned into the PPI is generated by function *test_pattern_generator ()*. The algorithm of the pseudorandom test pattern generator for PPI is shown below.

```
Static void PPI_Test_Pattern_Generator()
{
Catch current time as random generation seed;
Obtain the number of flip-flops in the circuit;
For (i=0;i<the number of flip-flops;i++)
{
Random number=rand (seed) %2;
Store random number in reset_value [] array;
}
}
```

This algorithm generates pseudorandom binary number 0 and 1, and stores those numbers in an array named *reset_value []*. Combining with function *is_statu_input (),* each PPI can have the new value when a new test pattern is applied to primary input.

After resetting the PPI, we turn the flip-flop back to the combinational circuit to become a regular sequential circuit, as shown in Fig. 6.3 (b). The fault injection will start in next step.

6.2.4 Fault List Generation

Fault list is used to identify the location where a stuck-at fault is inserted. The method of generating fault list is the same as the one described in Chapter 5. In addition, the function of generating fault list for sequential circuit automatically excludes lines of GND, VDD, RST and CLK. Furthermore, we deploy a counter associated with this function blocks so that the total number of faults on the list is recorded. The fault list generated for S27 circuit is shown as follow:

```
Generating fault list........
# Fault 0 list ---- G0
# Fault 1 list ---- G1
# Fault 2 list ---- G2
# Fault 3 list ---- G3
# Fault 4 list ---- G17
# Fault 5 list ---- G5
# Fault 6 list ---- G10
# Fault 7 list ---- G6
# Fault 8 list ---- G11
# Fault 9 list ---- G7
# Fault 10 list ---- G13
# Fault 11 list ---- G14
# Fault 12 list ---- G8
# Fault 13 list ---- G15
# Fault 14 list ---- G12
# Fault 15 list ---- G16
# Fault 16 list ---- G9
# ----------Fault list has been generated-------------
```

6.2.5 Fault Injection

Fault injection is the most significant part of the entire simulation. The general design of fault injection for sequential circuit is mostly the same as the one for combinational circuit. They both inject stuck-at faults into every single wire. The fault injection function sets all primary inputs, primary outputs and internal wires, including PPIs and PPOs (Pseudoprimary Output) but excluding GND, VDD, RST and CLK, with logical value 1 and 0, sequentially and respectively. The algorithm of fault injection has been mentioned in Section 5.2.5.

Due to the nature of sequential circuits, the fault injection needs to be partially upgraded. The reason is that, if we inject stuck-at fault into a line, the fault may not be observed in the

primary outputs in the first clock cycle. After the second clock the flip-flop receives a value, the fault then may be observed in the primary outputs in the third clock cycle as the values inside the sequential circuit are partially changed because of the flip-flop's effect. Fig. 6.4 gives an example of this situation.

If we apply test pattern 10011 to primary inputs and initialize PPI by 0 to this sequential circuit, the primary output K has 1 observed. If we insert SA0 in Line F, in the first clock cycle of 0→1, this fault cannot be observed. However, if we wait until the third clock cycle, which is 0→1 again, PPI becomes 1and faulty output in Line K is 0, compared with the correct output 1. This example strongly demonstrates the characteristic of fault injection of sequential circuits. Depending on the clock cycles, the flip-flop may be able to reveal the fault, enhancing the fault coverage.

Hence, if we setup the number of clock cycle properly, we can have higher fault coverage. But increasing the number of clock cycle, on the other hand, extends the whole simulation time. This is the tradeoff condition. In this thesis implementation, we initialize the number of clock cycle as 4, which contains two 0→1 positive clock cycles as well as two 1→0 negative clock cycles.

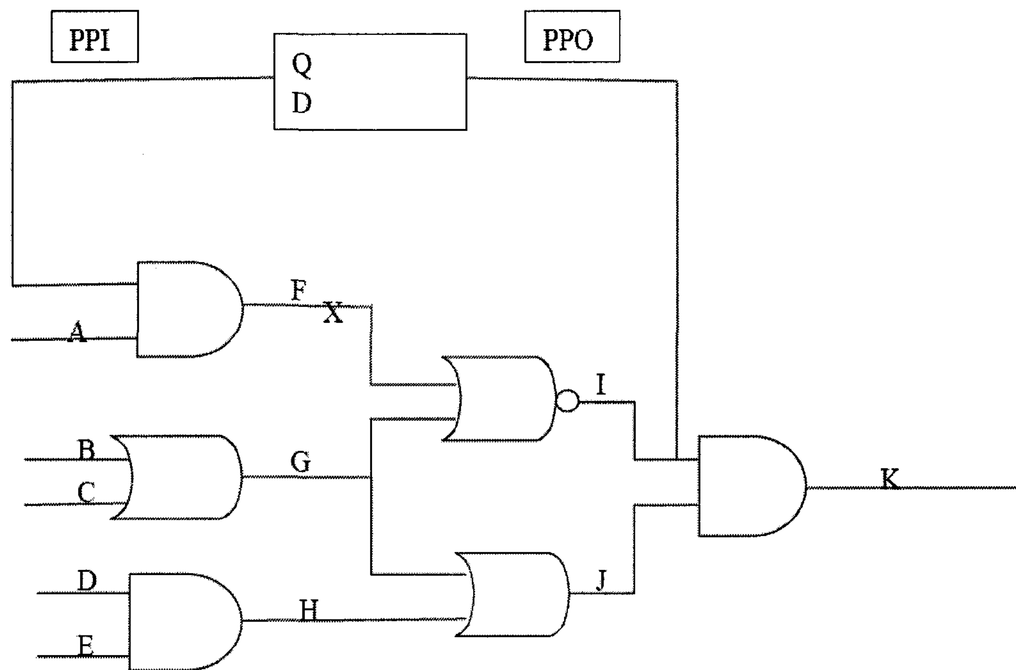

Figure 6.4 an example circuit with SA0 at Line F

In order to adapt this difference, the algorithm for fault injection for sequential circuit has been revised as follows:

```
Static void fault_injection()
{
if(Number of clock cycle<preset value)
{
Select a line to inject fault;
Switch(inject_fault_type)
{
Case: stuck-at-1
accForceFlag logical value 1 at selected line;
Case: stuck-at-0
accForceFlag logical value 0 at selected line;
}
Capture faulty output;
}
}
```

In this revised algorithm, the condition of injecting fault in a line becomes the number of clock cycle to drive the circuit to be simulated. The *accForceFlag* is still playing the same role as above in section 5.2.3. In addition, *accReleaseFlag* has been removed due to a fact that not only internal lines but also PPIs need to use this function. We thus build a special function block only for releasing the valued forced before to ease the whole fault injection.

6.2.6 Comparator Unit

In the forepart, we described the fault injection that inserts stuck-at faults to primary inputs, primary outputs and internal wires. For the sequential circuit, there are two instances of circuits, instance of faulty circuit and instance of fault free circuit, are running simultaneously. The comparator unit is responsible for comparing output from instance of faulty with the one from instance of fault free circuit. It conducts result comparison and

generates a fault report for the sequential circuit testing. The pseudocode description of the algorithm is given below.

```
Static void Comparator ()
{
Receive outputs from faulty instance and reference instance;
If (two outputs are different)
{
Switch (injection type)
{
Case: SA0
If this fault was not detected before, report this fault and remove it from fault list;
Case: SA1
If this fault was not detected before, report this fault and remove it fault list;
}
Report detected fault by this test pattern ;}
```

The report generated by this function indicates the number of detected faults by the test pattern, the fault coverage and the simulation time used. All the reports are saved in the fault "transcript. file" generated by ModelSim.

6.2.7 Testbench Design

After setting up PLI program in C/C++ environment, we turn our focus on the testbench design in ModelSim simulator. The testbench is capable of giving test data from PLI to the circuit, monitoring simulation process, and reporting the simulation results. It also instantiate the target benchmark circuits, and construct a bridge between PLI fault injection and benchmark circuits. The testbench design for sequential circuits is technically the same as the one for combinational circuits. However, the testbench for sequential circuits are required to setup two circuit instances instead of one. These two circuit instances have to run in the same clock cycle. A sample of testbench for S27 is shown as follow:

```
module tests27();
  wire [5:0]o;    reg[2:0]i; // output and input ports for fault free instance
  wire [5:0]o1;    reg[2:0]i1;// output and input ports for faulty instance
   reg[2:0]t; //input ports for GND,VDD, and CLK


   s27 s1(t[3],t[2],t[1],i1[2],i1[1],o1[5],o1[4],o1[3],o1[2],i1[0],o1[1],o1[0]);
   //instantiate S27 circuit as faulty instance
    s27 s(t[3],t[2],t[1],i[2],i[1],o[5],o[4],o[3],o[2],i[0],o[1],o[0]);
  //instantiate S27 circuit as fault free instance
 initial
 begin
 $Get_faultlist(s1);
 t=3'b111;i=3'b110;i1=3'b110; // initialize primary inputs of the circuit in ModelSim
end
always
 begin
 #1 i=$randomvector_d(s);   t[1]=$set_clock(s);// initialize primary inputs, PPIs, and
clock
i1=$randomvector_d(s1);  t[1]=$set_clock(s1);// use the same values as above
 #1 $Fault_injection;
   #1 $Get_output(o); //capture outputs from fault free instance
    $Get_Faulty_Output(o1);// capture outputs from faulty instance
 end
initial
 #($control) $stop; //receive control signal to control test cycles
Endmodule
```

In this testbench file, the details of the function used are given below:

$Get_faultlist(s1)$: responses to fault list generation of PLI, giving a complete set of nets of the circuit. This fault list is identical despite different instances from same circuit.

$randomvector(c): receives the test pattern from test pattern generation of PLI. This test pattern consists of a set of test pattern for primary inputs and a set of binary numbers for PPIs of this circuit.

$Fault_injection: corresponds to fault list generation of PLI to insert the stuck-at fault to each net of faulty instance. The function itself is able to distinguish which instance is for fault injection by identifying the name of instances (i.e. *s1* or *s*). As a result, it is not necessary to specify the name of faulty circuit in ModelSim.

$Get_output(o) and *$Get_Faulty_Output(o1)*: eventually capture the reference output and faulty output, and then send them to comparator unit of PLI to compare and composite the final results. We consider ModelSim simulator is good enough so that there is no propagation delay occurred.

Chapter 7

Test Case and Experimental Results

In this chapter, we discuss fault injection simulation experiments on both ISCAS 85 combinational benchmark circuits and ISCAS 89 full-scan sequential benchmark circuits based on our PLI fault injection approach within ModelSim commercial simulator. Many of the details of implementation will be provided below considering simple benchmark circuits. We also compare and discuss our results with existing works.

7.1 Pseudorandom Test Pattern Generation

a) Combinational circuit

The initial step of fault injection simulation is to generate the test pattern. We take c17 benchmark circuit as an example. The test pattern generation function block of PLI is trigged. The output of these test patterns is shown below.

```
# Input vector ---  01001   Fault free output:    11
# Input vector ---  00011   Fault free output:    01
# Input vector ---  11110   Fault free output:    10
# Input vector ---  00100   Fault free output:    00
# Input vector ---  00001   Fault free output:    01
# Input vector ---  10000   Fault free output:    00
# Input vector ---  10001   Fault free output:    01
```

In the meantime, these generated test patterns are directly sent to CUT, and then we can have the fault free output of this circuit.

b) Sequential circuit

For sequential circuit, we take s298 benchmark circuit as an example. The additional function of test pattern generation is to assign the value to PPIs. Moreover, there is no fault free output reported for sequential circuit due to the fact that we are using reference output instead of fault free output. Besides, the reference circuit receives the same test patterns. The test patterns for primary inputs are shown as follow.

```
# Input vector --- 010
# Input vector --- 101
# Input vector --- 000
# Input vector --- 100
# Input vector --- 111
```

7.2 Fault list generation

Fault list is generated to provide the name of each line inside the circuit and calculate the total number of stuck-at faults of the circuit. The function in sequential circuit is identical. The result of fault list of c17 is displayed below.

```
# Generating fault list........
# Fault 0  list ----  N1
# Fault 1  list ----  N2
# Fault 2  list ----  N3
# Fault 3  list ----  N6
# Fault 4  list ----  N7
# Fault 5  list ----  N22
# Fault 6  list ----  N23
# Fault 7  list ----  N10
# Fault 8  list ----  N11
# Fault 9  list ----  N16
# Fault 10  list ----  N19
# ----------Fault list has been generated-------------
```

7.3 Fault injection simulation

After preparing fault free output and fault list for the simulation, the program is starting fault injection. Each line is set by logical value 0 and 1, respectively. The results are reported in "transcript. file". The content of this file of c17 is shown below.

```
# Generating fault list........
# Fault 0  list ----  N1
# Fault 1  list ----  N2
# Fault 2  list ----  N3
# Fault 3  list ----  N6
# Fault 4  list ----  N7
# Fault 5  list ----  N22
# Fault 6  list ----  N23
# Fault 7  list ----  N10
# Fault 8  list ----  N11
# Fault 9  list ----  N16
# Fault 10  list ----  N19
# ----------Fault list has been generated-------------
# Fault free output:    00
# Input vector ---  01001   Fault free output:    11
# Input: 01001 Output: 01 net N2 stuck-at-0 detected
# Input: 01001 Output: 01 net N22 stuck-at-0 detected
# Input: 01001 Output: 10 net N23 stuck-at-0 detected
# Input: 01001 Output: 00 net N11 stuck-at-0 detected
```

```
# Input: 01001 Output: 01 net N16 stuck-at-1 detected
#
#  Fault Coverage ------------  22.727273 %
# Time used :  0.000000 second
# 1 test vectors used
#
# =================Starting new test vector=====================
# Input vector ---  00011   Fault free output:    01
# Input: 00011 Output: 00 net N7 stuck-at-0 detected
# Input: 00011 Output: 11 net N10 stuck-at-0 detected
# Input: 00011 Output: 11 net N16 stuck-at-0 detected
# Input: 00011 Output: 11 net N2 stuck-at-1 detected
# Input: 00011 Output: 00 net N3 stuck-at-1 detected
# Input: 00011 Output: 11 net N22 stuck-at-1 detected
# Input: 00011 Output: 00 net N19 stuck-at-1 detected
#
#  Fault Coverage ------------  54.545455 %
# Time used :  0.000000 second
# 2 test vectors used
#
# =================Starting new test vector=====================
# Input vector ---  11110   Fault free output:    10
# Input: 11110 Output: 00 net N1 stuck-at-0 detected
# Input: 11110 Output: 11 net N3 stuck-at-0 detected
# Input: 11110 Output: 11 net N6 stuck-at-0 detected
# Input: 11110 Output: 11 net N19 stuck-at-0 detected
# Input: 11110 Output: 11 net N23 stuck-at-1 detected
# Input: 11110 Output: 00 net N10 stuck-at-1 detected
# Input: 11110 Output: 11 net N11 stuck-at-1 detected
#
#  Fault Coverage ------------  86.363636 %
# Time used :  0.000000 second
# 3 test vectors used
#
# =================Starting new test vector=====================
# Input vector ---  00100   Fault free output:    00
# Input: 00100 Output: 10 net N1 stuck-at-1 detected
# Input: 00100 Output: 01 net N7 stuck-at-1 detected
#
#  Fault Coverage ------------  95.454545 %
# Time used :  1.000000 second
# 4 test vectors used
#
# =================Starting new test vector=====================
# Input vector ---  00001   Fault free output:    01
#
#  Fault Coverage ------------  95.454545 %
# Time used :  1.000000 second
# 5 test vectors used
#
# =================Starting new test vector=====================
# Input vector ---  01100   Fault free output:    11
# Input: 01100 Output: 00 net N6 stuck-at-1 detected
#
#  Fault Coverage ------------  100.000000 %
```

```
# Time used :  1.000000 second
# 6 test vectors used
```

The results include the input test patterns, the name of the line, the fault detected by the test pattern, and the simulation time.

7.4 Test Results and Analysis of Combinational Circuits

The PLI fault injection simulation has been implemented in C language associated with ModelSim simulator. Various test patterns have been generated for these ISCAS 85 benchmark circuits to evaluate this proposed approach.

Simulations were carried on 10 ISCAS 85 benchmark circuits. Detailed description of each circuit is given in Table 7.1.

Circuit name	# of inputs	# of outputs	# of wires	# of gates
c17	5	2	11	6
c432	36	7	196	160
c499	41	32	243	202
c880	60	26	443	383
C1355	41	32	587	497
c1908	33	25	913	880
c2670	233	140	1052	1269
c3540	50	20	1719	1669
C5315	178	123	2485	2307
C6288	32	32	2448	2416
C7552	207	108	3720	3513

Table 7.1 Combinational circuit description

Based on the description of each circuit, we apply different number of test patterns to each circuit in a reasonable time frame in order to have good fault coverage. The results are shown in Table 7.2.

Circuit name	No. of faults	No. of detected faults	test patterns applied	fault coverage	simulation time(sec)
c17	22	22	6	100.00%	1
c432	392	389	355	99.23%	34
c499	486	482	1329	99.18%	180
c880	886	867	1982	97.86%	895
c1355	1174	1147	928	97.70%	588
c1908	1826	1794	1328	98.25%	1350
c2670	2104	1753	309	83.32%	902
c3540	3438	3015	462	87.70%	3567
c5315	4970	4774	472	96.06%	3074
c6288	4896	4863	273	99.33%	1964
c7552	7440	7010	630	94.22%	9524

Table 7.2 Results of fault injection by PLI approach

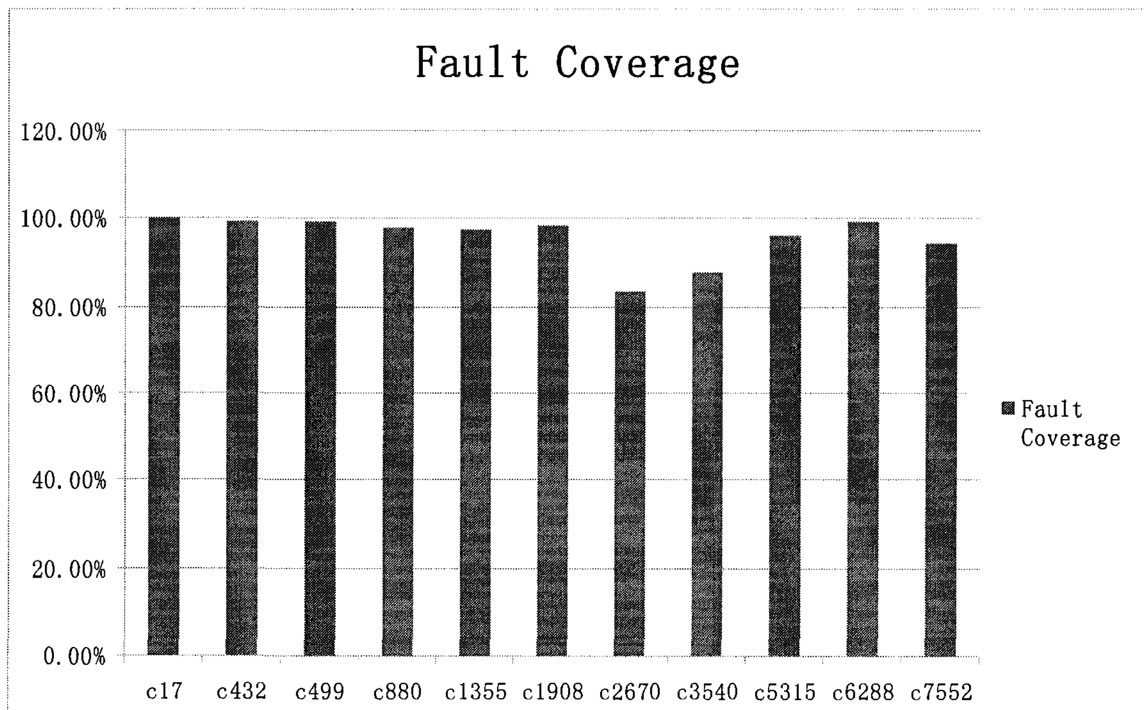

Figure 7.1 Fault coverage of ISCAS 85 benchmark circuits by PLI approach

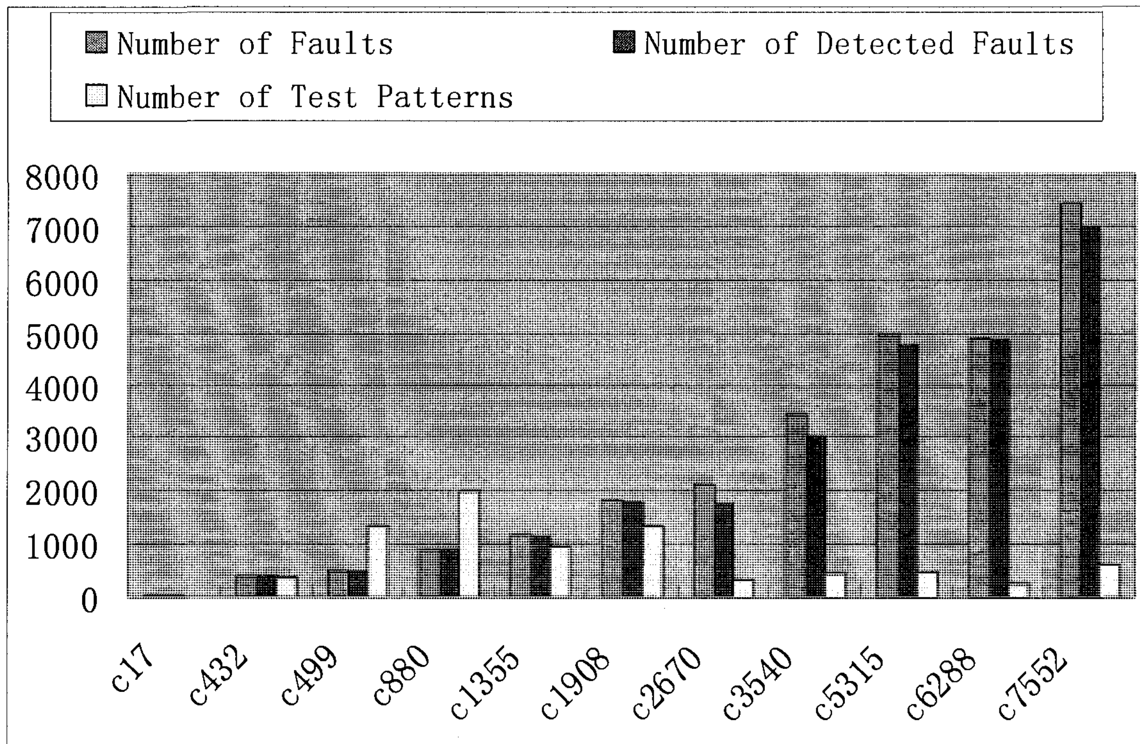

Figure 7.2 Detected faults vs. total faults, and test patterns

From Table 7.2 above, we can see that simulation time is increased depending highly on the complexity of circuit tested. Moreover, fault coverage is very close to 100% when numerous test patterns are applied, regardless of the complexity of circuit tested. However, we found that some faults are not detectable, no matter how many test patterns we apply. For example, c432 has four undetectable faults, even though we apply thousands of pseudorandom test patterns. FSIM also reports these undetectable faults.

As we mentioned above, our approach is very similar with the one proposed in [8]. We both apply circuits in the form of Verilog as the hardware prototype, and use commercial simulators (Max Plus II vs. ModelSim). The comparison between two approaches is shown below in Table 7.3.

Circuit name	# of test patterns	Fault Coverage	Simulation time(sec)	Fault Coverage from [8]	Simulation time(sec)
c17	N/A	100.00%	1	100.00%	8
c432	199	98.21%	20	95.92%	430
c499	199	94.65%	30	88.89%	411
c880	199	93.45%	94	92.55%	1150
c1355	199	94.38%	127	87.22%	1543
c1908	199	90.75%	208	83.57%	3729
c2670	199	82.66%	593	N/A	N/A
c3540	199	82.58%	1260	N/A	N/A
c5315	199	94.45%	1321	97.99%	41559
c6288	199	99.10%	1457	99.31%	59308
c7552	199	91.22%	3048	N/A	N/A

Table 7.3 Results comparison

From Table 7.3, we can see that fault coverage and simulation time have been improved, compared with previous work in [8]. Previous approach has to compile the faulty circuit every time after a fault is injected. In the meantime, the fault free outputs are stored in a file. The comparison unit has to open and read this file to achieve the task. Our approach has improved these drawbacks. The circuits in the format of Verilog and fault injector are compiled just once. The fault free outputs are stored in memory of computer, where it is much faster to access.

7.5 Test Results and Analysis of Sequential Circuits

Table 7.4 is giving detailed description of sequential circuits ISCAS'89. Based on this description, we apply pseudorandom test patterns to the circuits. Due to our hardware limitation, for some very complex sequential circuits, we just apply a few pseudorandom test patterns. The fault coverage should be higher if more test patterns are applied in future. The results are shown in Table 7.5, Fig. 7.3 and Fig. 7.4.

circuit name	No.of DFFs	No.of gates	No.of inverters	No.of inputs (exluding clock and reset)	No. of outputs	No. of wires
s298	14	75	44	3	6	136
s344	15	101	59	9	11	184
s349	15	104	57	9	11	185
s382	21	99	59	3	6	182
s400	21	106	58	3	6	188
s444	21	119	62	3	6	205
s526	21	141	52	3	6	217
s641	19	107	272	35	24	433
s713	19	139	254	35	23	447
s832	5	262	25	18	19	310
s953	29	311	84	16	23	440
s1196	18	388	141	14	14	561
s1238	18	428	80	14	14	540
s1423	74	490	167	17	5	748
s1488	6	550	103	8	19	667
s5378	179	1004	1775	35	49	2993
s15850	534	3448	6324	77	150	10383
s35932	1728	12204	3861	35	320	17828

Table 7.4 Sequential circuit description

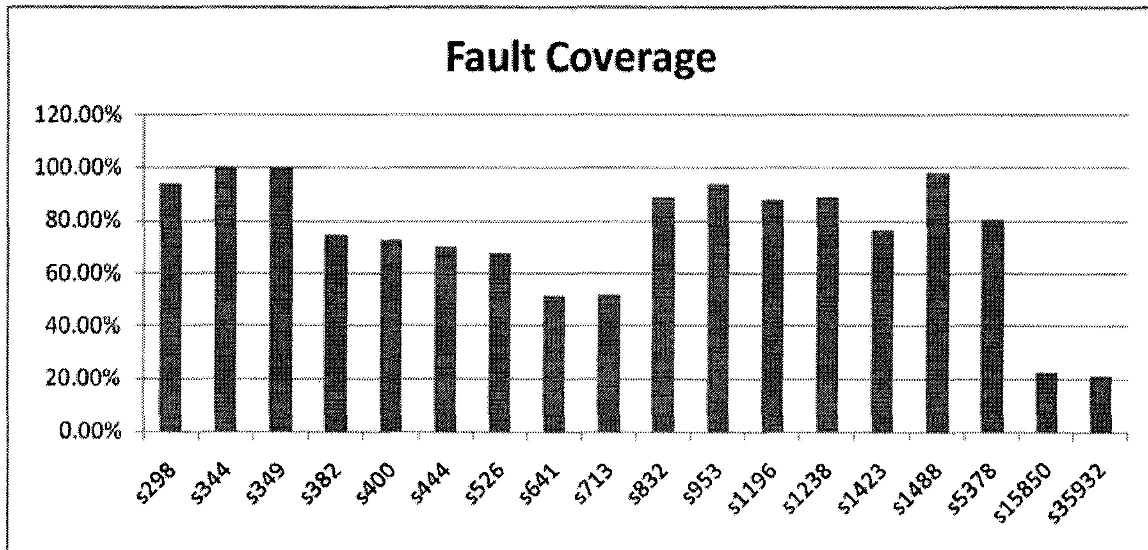

Figure 7.3 Fault coverage of ISCAS 89 benchmark circuits by PLI

circuit name	flip-flops	Fault coverage	Simulation time (sec)	test patterns applied	No.of faults simulated	No.of detected faults
s298	14	94.12%	847	1572	272	256
s344	15	100.00%	301	306	368	368
s349	15	100.00%	174	173	370	370
s382	21	74.73%	852	888	364	272
s400	21	72.87%	618	576	376	274
s444	21	69.76%	827	685	410	286
s526	21	67.74%	1123	847	434	294
s641	19	51.50%	2603	521	866	446
s713	19	51.68%	3084	405	894	462
s832	5	89.03%	4632	1586	620	552
s953	29	94.09%	3483	445	880	828
s1196	18	87.88%	8234	911	1122	986
s1238	18	89.07%	10175	919	1080	962
s1423	74	76.60%	13508	777	1496	1146
s1488	6	98.20%	8924	664	1334	1310
s5378	179	80.49%	57381	201	5986	4818
s15850	534	22.04%	225492	62	20766	4577
s35932	1728	21.07%	43635	4	35656	7513

Table 7.5 Results of fault coverage by PLI fault injection

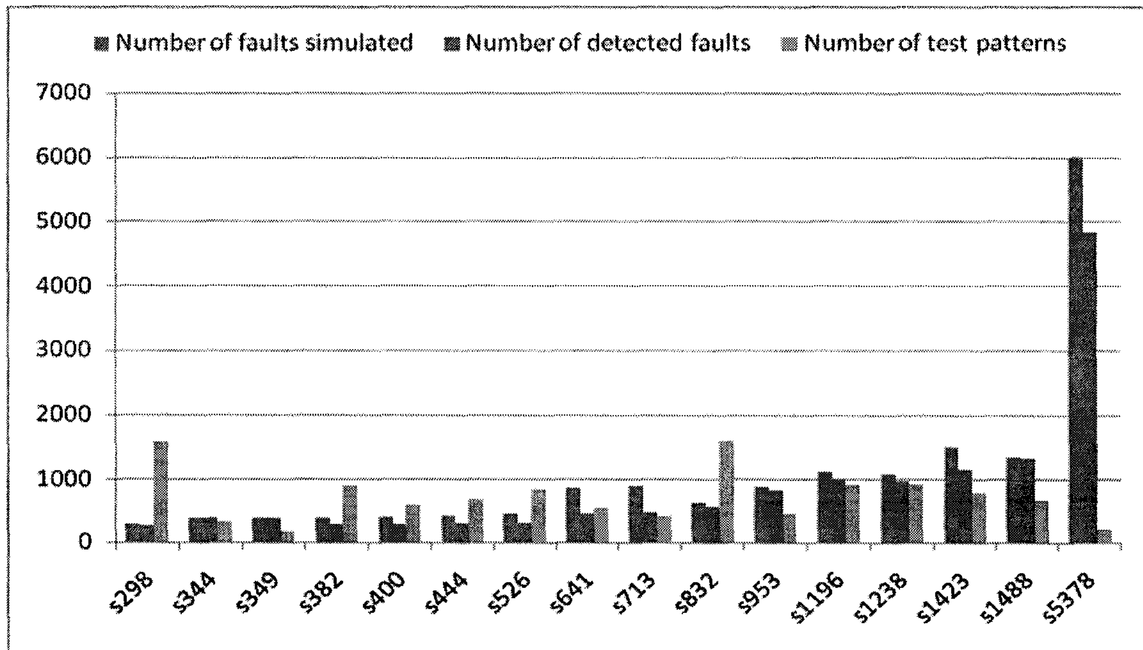

Figure 7.4 Number of faults simulated vs. Number of detected faults and test patterns

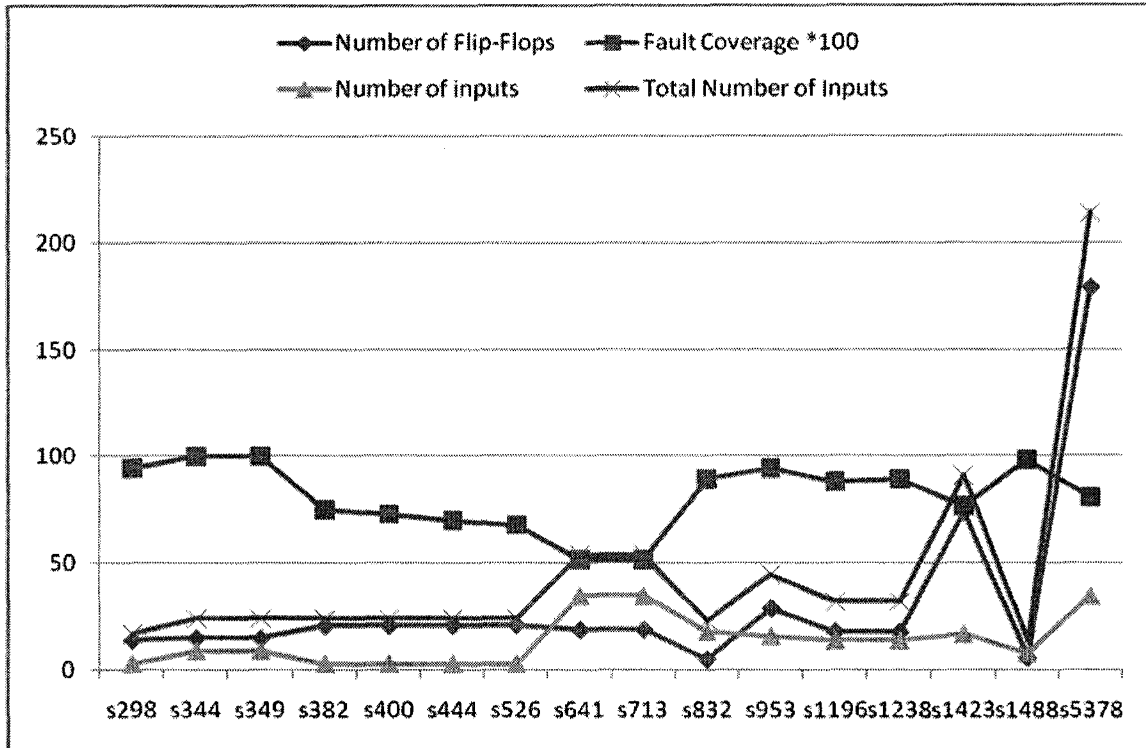

Figure 7.5 Number of inputs vs. fault coverage in each sequential circuits

From Fig.7.5 above, we can see that the number of flip-flops and primary inputs becomes one of important factors to fault coverage in our fault injection approach. The reason is that, when applying a new test pattern to primary inputs, we also assign random binary values to PPIs. Therefore, it is very important for this approach to combine proper test patterns in primary inputs with initial values in PPIs. The posibility of generating a proper combination of primary inputs and PPIs is lower if either the number of primary inputs or the number of PPIs is large. In the case of some sequential circuits with the large number of PPIs, more pseudorandome test patterns are needed to generate the proper values for PPIs. Hence the fault coverage partially relies on this posibility especially when the pseudorandom test patterns applied are limited.

Chapter 8

Conclusions

A Verilog HDL-based fault injection approach is proposed in the thesis to detect single Stuck-At faults. This improved approach synthesizes the PLI function of Verilog HDL and a commercial simulator ModelSim for both combinational and sequential circuits. The essential idea of our fault simulator is designed for testing the circuits that are described in Verilog HDL. A typical BIST environment is setup with pseudorandom test pattern generator that sends its output to feed the CUT (circuit under test). The output streams of CUT are sent to comparator unit in which a fault is detected if it is different from the fault-free output. Compared with previous work, this fault injection approach is well improved in the test efficiency in terms of fault coverage and simulation time. All gate-level hardware in the form of Verilog can achieve stuck-at fault injection by directly setting logic 1 or 0 in the target line of hardware. Besides, this fault injection offers a very flexible test environment, where the ModelSim simulator used in this thesis can easily be replaced by any other Verilog hardware simulation and our approach still can be achieved.

Furthermore, this approach enables fault injection to combinational circuits and sequential circuits without significantly modifying infrastructure code. Not only can it simply insert faults to each combinational circuit, but also it can deal with the complex sequential circuits with numerous flip-flops. The reference circuit is created to overcome the nature of multiple states, which is the most difficult part of sequential circuits testing. It also reduces the simulation memory used. In some conventional sequential circuit testing approach, a set of fault-free output still has to be stored before injection. This requires sufficient simulation memory depending on the complexity of CUT. Therefore, associated reference circuit used can avoid this drawback, and then on the other hand, less required simulation memory can reduce the testing cost.

The entire simulation process is completely automatic, and no intervention is required during the process after the environment is set. The detailed architecture and applications of

this fault injection approach are also described in this thesis. Results of benchmark circuits ISCAS85 and ISCAS89 are provided as well.

We hope that this research can further lead more works on this domain. The future research will mainly include two categories: the enhancement of ATPG for this co-design fault injection approach, and improvement of procedure of fault simulation in ModelSim simulator.

Although ATPG is a well-known general test pattern generation algorithm, it still has to be modified to adapt this software and hardware co-design fault injection approach. With some limited conditions, the detailed structure inside of the circuit tested cannot be well understood by the program. Some ATPG algorithm such as PODEM cannot be applied properly. Therefore, a new approach needs to be created to perform ATPG for this co-design environment such that based on the number of AND gates and OR gates and interconnection among them the test pattern in primary inputs can be determined. It would be the active deterministic test pattern generation that does not need back trace and forward propagation from fault injected location of circuit to verify the assigned input value.

On the other hand, the effort should be given to parse the circuit by applying software application. As we see in previous section, simulation time becomes an issue when the complexity of circuit increases, especially for sequential circuits. This issue can be mostly solved by reducing the number of faults simulated. If we can deeply parse the circuit and realize the interconnection and routing of fault propagation, we will be able to eliminate multiple faults by one test pattern such that if a fault is detected, those wires whose logical values become opposite during the routing of fault propagation can be considered as a Stuck-at fault injected. According to this principle, a bunch of faults can be eliminated after one fault is detected. By shortening fault list for simulation, the final simulation time will be dramatically reduced.

References

[1] W. Wolf, Modern VLSI Design: System-on-Chip Design, Prentice-Hall, 2002

[2] N.K. Jha and S. Gupta, Testing of Digital Systems, Cambridge University Press, 2003

[3] C. Hawkins, H.T. Nagle, R. Fritzemeier and J. Guth, The VLSI Circuit Test Problem-A Tutorial, IEEE Trans. Industrial Electronics, Vol. 36, No. 2, pp. 111-116, 1989

[4] P. Goel, Test Generation Costs Analysis and Projections, in Proc. of 17th Design Automation Conference, pp. 77-84, 1980

[5] H.J. Wunderlich, BIST For Systems-On-A-Chip, Integration, the VLSI Journal, Vol. 26, Issue 1-2, pp. 55-78, 1998

[6] S.R. Das, Getting Errors to Catch Themselves---Self-Test of VLSI Circuits With Built-In Hardware, IEEE Trans. on Instrumentation and Measurement, Vol. 54, No. 3, pp. 941-955, 2005

[7] Z. Navabi, Verilog Digital System Design, Second Edition, McGraw-Hill, 2006

[8] M.H. Assaf, S.R. Das, E.M. Petriul, L. Jin, C. Jin, D. Biswas, V. Groza and M. Sahinoglu, Hardware and Software Co-Design in Space Compaction of Cores-Based Digital Circuits, Instrumentation and Measurement Technology Conference, pp. 1503-1508, 2004

[9] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs and G. H.Leber, Comparison of Physical and Software-Implemented Fault Injection Techniques, IEEE Trans. on Computers, Vol. 52, Issue 9, pp. 1115-1133, 2003

[10] C. Jin, Test Implementation of Embedded Cores-Based Sequential Circuits Using Verilog HDL under Altera MAX PLUS II Development Environment, Master Thesis, University of Ottawa, 2004

[11] S.R. Das, S. Mukherjee, E.M. Petriu, M.H. Assaf, M. Sahinoglu and W.B. Jone, An Improved Fault Simulation Approach Based on Verilog With Application to ISCAS Benchmark Circuits, in Proc. of Instrumentation and Measurement Technology Conference, pp. 1902-1907, 2006

[12] M.E. Hsueh, T.K. Tsai and R.K. Iyer, Fault Injection Techniques and Tools, Computer, Vol. 30, Issue 4, pp. 75-82, 1997

[13] S. Mittra, Principles of Verilog PLI, Kluwer Academic Publishers, 1999

[14] N.A. Kanawati, G.A. Kanawati and J.A. Abraham, Dependability Evaluation using Hybrid Fault /Error Injection, in Proc. of Computer Performance and Dependability Symposium, pp. 224-233, 1995

[15] B.Brown, Designing Combinational Circuits, Retrieved on 2007/10/5, www.spsu.edu/cs/faculty/bbrown/web_lectures/postfix

[16] G.A. Kanawati, N.A. Kanawati and J.A. Abraham, FERRARI:A Flexible Software-Based Fault and Error Injection System, IEEE Trans. on Computers, Vol. 44, No. 2, pp. 248-260, 1995

[17] J.A. Clark and D.K. Pradhan, Fault Injection, Computers, Vol. 28, No. 6, pp. 47-56, 1995

[18] C.I.H. Chen and S. Perumal, Analysis of The Gap Between Behavioral And Gate Level Fault Simulation, in Proc. of 6th Annual IEEE International ASIC Conference and Exhibit, pp. 144-147, 1993

[19] M. H. Assaf, R. S. Abielmona, P. Aboulghasem, S. R. Das, E. M. Petriu, V. Groza, and M. Sahinoglu, Test Implementation of Embedded Cores-Based Digital Devices in JBits Java Simulation Environment, in Proc. of 7th International Conference on Information Technology, pp. 315-325, 2004

[20] ISCAS 85 and ISCAS 89 benchmark circuits: http://www.fm.vslib.cz/~kes/asic/iscas/

[21] T.A. Delong, B.W. Johnson and J.A. Profeta, A Fault Injection Technique for VHDL Behavioral-Level Models, IEEE Design and Test of Computers, Vol. 13, No. 4, pp. 24-33, 1996

[22] R.J. Hayne and B.W. Johnosn, Behavioral Fault Modeling in a VHDL Synthesis Environment, in Proc. of 17th IEEE VLSI Test Symposium, pp. 333-340, 1999

[23] S. Han, K.G. Shin and H.A. Rosenberg, DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-time Systems, in Proc. of International Computer Performance and Dependability Symposium, pp. 204-213, 1995

[24] R.R. Some, W.S. Kim, G. Khanoyan, L. Callum, A. Agrawal and J. Beahan, A Software-Implemented Fault Injection Methodology for Design and Validation of System Fault Tolerance, in Proc. of International Conference on Dependable Systems and Networks, pp. 501-506, 2001

[25] T.R. Padmanabhan and B.B.T. Sundari, Design Through Verilog HDL, Wiley-Interscience Press, 2004

[26] H.R. Zarandi, S.G. Miremadi and A. Ejlali, Dependability Analysis Using a Fault Injection Tool Based on Synthesizability of HDL Models, in Proc. of 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 485-492, 2003

[27] J.C. Baraza, J. Gracia, D. Gil and P.J. Gil, Improvement of Fault Injection Techniques Based on VHDL Code Modification, in Proc. of 10th IEEE International High-Level Design Validation and Test Workshop, pp. 19-26, 2005

[28] R. Reis, M. Lubaszewski and J.A.G. Jess, Design of Systems on a Chip: Design and Test, Springer Press, 2006

[29] H.K. Lee and D.S. Ha, New Methods of Improving Parallel Fault Simulation in Synchronous Sequential Circuits, in Proc. of IEEE International Conference on Computer-aided Design, pp. 10-17, 1993

[30] D.G. Saab, Parallel-Concurrent Fault Simulation, IEEE Trans. on Very Large Scale Integration Systems, Vol. 1, No. 3, pp. 356-363, 1993

[31] M. Phadoongsidhi, K.T. Le and K.K. Saluja, A Concurrent Fault Simulation for CrossTalk Faults in Sequential Circuits, in Proc. of 11th Asian Test Symposium, pp. 182-187, 2003

[32] P.A. Riahi, A. Navabi and F. Lombardi, Simulating Faults of Combinational IP Core-based SOCs in a PLI Environment, in Proc. of 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 389-397, 2005

[33] S. Seshu, On an Improved Diagnosis Program, IEEE Trans. on Electronic Computers, Vol. EC-14, pp. 567-580, 1967

[34] A. Miczo, Digital Logic Testing and Simulation, 2nd Edition, Wiley-Interscience Press, 2006

[35] F. Celeiro, L. Dias, J. Ferreira, M.B. Santos and J.P. Teixeira, Defect-Oriented IC Test and Diagnosis Using VHDL Fault Simulation, in Proc. of International Test Conference, pp. 620-628, 1996

[36] C.A. Ryan and J.G. Tront, VHDL Switch-Level Fault Simulation, ,in Proc. of the Conference on European Design Automation, pp. 650-655, 1994

[37] M.B. Santos and J.P. Teixeira, Defect-Oriented Mixed-Level Fault Simulation of Digital Systems-on-a-Chip Using HDL, in Proc. of the Conference on Design, Automation and Test in Europe, pp. 549-554, 1999

[38] J. Arlat, M. Agurera, L. Amat, Y. Crouzet, J.C. Fabre, J.C. Laprie, E. Martins and D. Powell, Fault Injection for Dependability Validation: A Methodology and Some Applications, IEEE Trans. on Software Engineering, Vol. 16, Issue 2, pp. 166-182, 1990

[39] A. Fin and F. Fummi, A VHDL Simulator for Functional Test Generation, in Proc. of Design, Automation and Test in Europe Conference, pp. 390-395, 2000

[40] W. Kao and R.K. Iyer, DEFINE:A Distributed Fault Injection and Monitoring Environment, in Proc. of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, pp. 252-259, 1994

[41] T.R. Padmanabhan and B.B.T. Sundari, Design Through Verilog HDL, Wiley-Interscience Press, 2004

[42] D. Kay and S. Mourad, Controllable LFSR For BIST, in Proc. of 17th IEEE Instrumentation and Measurement Technology Conference, pp. 223-228, 2000

[43] O. Gunnetlo, J. Karlsson, and J. Tonn, Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation, in Proc. of 19th Annual. International Symposium on Fault-Tolerant Computing, pp. 340-347, 1989

[44] M. Elbadri, V. Groza, R. Abielmona and M. Assaf, A Reconfigurable Processing Unit for Digital Circuit Testing Using Built-In Self-Test Techniques, in Proc. of Instrumentation and Measurement Technology Conference, pp. 239-244, 2006

[45] J. Karlsson, J. Arlat, and G. Leber, Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture, in Proc. of 5th IEEE International Working Conference on Dependable Computing for Critical Applications, pp. 150-161, 1995

[46] S.R. Das, C. Jin, L. Jin, M. Assaf, E.M. Petriu and M. Sahinoglu, Altera Max Plus II Development Environment in Fault Simulation and Test Implementation of Embedded Cores-Based Sequential Circuits, in Proc. of 6th International Workshop on Distributed Computing, 2004

[47] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson and J. Karlsson, Fault Injection into VHDL Models: The MEFISTO Tool, in Proc. of 24th International Symposium on Fault Tolerant Computing, pp. 66-75, 1994

[48] A. Steiniinger, B. Rahbaran and T. Handl, Built-in Fault Injectors-The Logical Continuation of BIST?, in Proc. of 1st Workshop on Intelligent Solutions in Embedded Systems, pp. 327-332, 2003

[49] J. Guthoff and V. Sieh, Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method, in Proc. of 25th International Symposium on Fault-Tolerant Computing, pp. 196-206, 1995

[50] F. Maamari, and J. Rajski, The Dynamic Reduction of Fault Simulation, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol. 12, No. 1, pp. 137-148, 1993

[51] S.R. Das, H.T. Ho, W.B. Jone and A.R. Nayak, An Improved Output Compaction Technique for Built-In Self-Test in VLSI Circuit, in Proc. of 8th International Conference on VLSI Design, pp. 403-407, 1995

[52] S.R. Das, C.V. Ramamoorthy, M.H. Assaf, E.M. Petriu, W.B. Jone and M. Sahinoglu, Fault Simulation and Response Compaction in Full Scan Circuits Using HOPE, IEEE Trans. on Instrumentation and Measurement, Vol. 54, No. 6, pp. 607-612, 2005

[53] H.K. Lee and D.S. Ha, An Efficient Forward Fault Simulation Algorithm Based On the Parallel Pattern Single Fault Propagation, in Proc. of the IEEE International Test Conference on Test, pp. 946-955, 1991

[54] I. Pomeranz, L.N. Reddy and S.M. Reddy, COMPACTEST: A Method To Generate Compact Test Sets for Combinational Circuits, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol. 12, Issue 7, pp. 1040-1049, 1993

[55] C. Dawson, S.K. Pattanam and D. Roberts, The Verilog Procedural Interface for the Verilog Hardware Description Language, in Proc. of IEEE International Verilog HDL Conference , pp.17-23, 1996

[56] D.J. Smith, VHDL &Verilog Compared &Contrasted-Plus Modeled Example Written in VHDL, Verilog and C, in Proc. of 33rd Design Automation Conference, pp. 771-776, 1996

[57] P. Goel, An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits, IEEE Trans. on Computers, Vol. C-30, No. 3, pp. 215-222, 1981

[58] H.K. Lee and D.S. Ha, HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol. 15, No. 9, pp. 1048-1058, 1996

[59] T.M. Niermann, W.T. Chen, and J.H. Patel, PROOFS: A Fast, Memory-Efficient Sequential Circuit Fault Simulator, IEEE Trans. on Computer-Aided Design, Vol. 11, No. 2, pp. 198-207, 1992