Université d'Ottawa · University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Chuan JIN

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

M. Sc. (Systems Science)

GRADE - DEGREE

Systems Science

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

Test Implementation of Embedded Cores-Based Sequential Circuits Using Verilog HLD Under Altera Max Plus II Development Environment

S. Das

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

E. Petriu

CO-DIRECTEUR DE LA THÈSE - THESIS CO-SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

V. Groza

A. Nayak

J-M. De Koninck, Ph D

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

DEAN OF THE FACULTY OF GRADUATE AND POSTODORAL STUDIES

TEST IMPLEMENTATION OF EMBEDDED CORES-BASED SEQUENTIAL CIRCUITS USING VERILOG HDL UNDER ALTERA MAX PLUS II DEVELOPMENT ENVIRONMENT

CHUAN JIN

THESIS SUBMITTED TO THE

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF SCIENCE IN SYSTEMS SCIENCE

SYSTEMS SCIENCE

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

UNIVERSITY OF OTTAWA

© CHUAN JIN, OTTAWA, CANADA, 2004

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: 0-494-01504-7
Our file Notre référence
ISBN: 0-494-01504-7

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.

ACKNOWLEDGMENTS

I would like to express my sincerest appreciation and gratitude 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, constant support, both moral and technical, and active cooperation during the entire period this research was carried out. Without him this work would never have materialized in its current form.

I would also like to sincerely thank my thesis co-supervisor Dr. Emil M. Petriu, Professor, School of Information Technology and Engineering, University of Ottawa.

Thanks are also due to Dr. Mansour H. Assaf of the School of Information Technology and Engineering, University of Ottawa, for his constant help and unforgettable support during the entire period of this research.

I would also like to express my gratitude to all my friends and colleagues at the University of Ottawa, particularly Mr Liwu Jin, who helped me immensely in successful completion of this work.

Finally, I owe very special thanks and gratitude to my wife and my parents for their love and never-ending support.

TABLE OF CONTENTS

ACKNOWLEDGMENTS i

TABLE OF CONTENTS ii

LIST OF TABLES v

LIST OT FIGURES vi

LIST OF ACRONYMS vii

ABSTRACT ix

CHAPTER 1 INTRODUCTION 1

1.1. Digital Circuits Testing – Basic Concepts 1

1.2. Testing Problem 3

1.3. Test Generation Techniques 4

1.4. Organization of the Thesis 7

CHAPTER 2 LOGIC SIMULATION AND FAULT ANALYSIS 8

2.1. Simulation 8

2.1.1. Circuit modeling 8

2.1.2. Signals 9

2.1.3. Modeling of delays 9

2.1.4. Simulator 10

2.1.5. Event-driven simulation 11

2.2. Fault Modeling 12

2.2.1. Stuck faults 13

2.2.2. Fault collapsing 13

2.2.3. Other fault models 15

2.3. Fault Simulation 16

2.3.1. Methods of fault simulation 17

2.3.2. A comparison of fault simulation methods 19

2.4. Fault Simulation Tools 21

2.5. Fault Coverage and Product Quality 22

CHAPTER 3 TEST GENERATION APPROACHES 24

3.1. Test Generation for Sequential Circuits 25

3.2. Approaches for Test Generation 25

3.2.1. Iterative array approach 25

3.2.1.1. Extended D-algorithm 27

3.2.1.2. Nine-value algorithm 28

3.2.1.3. SOFTG 29

3.2.1.4. Backtrace algorithms 29

3.2.2. Verification-based approaches 30

3.2.2.1. SCIRTSS 30

3.2.3. Functional and expert system approaches 31

CHAPTER 4 FAULT SIMULATION UNDER ALTERA MAX PLUS II DESIGN ENVIRONMENT 32

4.1. Designing Sequential Circuit with Verilog HDL 32

4.1.1. Developing Verilog hardware description language 32

4.1.2. Digital circuit design and Verilog HDL 33

4.1.3. Design styles and abstraction levels in Verilog HDL 35

4.1.4. An example: s27 benchmark circuit described by Verilog HDL 36

4.2. Simulation Under MAX PLUS II Version 10.1 39

4.2.1. Altera MAX PLUS II design environment 41

4.2.2. Simulation flow under Altera MAX PLUS II 42

4.2.3. MAX PLUS II applications 43

CHARPTER 5 DESIGN AND IMPLEMENTATION 46

5.1. General Structure in Sequential Circuit Testing 46

5.2. Initialization 48

5.2.1. Random Signal Generator 48

5.2.1.1. Hardware implementation of random signal generator 48

5.2.1.2. Software simulation of random signal generator 49

5.2.2. Input vector file 50
5.3. Fault Injection and Fault Simulation 53
5.3.1. Hardware fault injection 53
5.3.2. Software fault injection 54
5.3.2.1. Input part fault simulation 54
5.3.2.2. Inside wire fault simulation 55
5.3.2.3. Output part fault simulation 57
5.4. Comparison of Results and Reports 58

CHAPTER 6 TEST CASES AND EXPERIMENTAL RESULTS 62
6.1. Generating Random Signal Input Vector File of s298 Circuit 62
6.2. Generating Fault-Free Output File of s298 Circuit 63
6.3. Fault Injection and Simulation for s298 Circuit 65
6.4. Fault Detection Report on s298 Benchmark Circuit Testing 67
6.5. Test Results and Analysis of ISCAS 89 Sequential Benchmark Circuits 69

CHAPTER 7 CONCLUSIONS 74

REFERENCES 76

LIST OF PAPERS BY THE CANDIDATE 81

APPENDIX 82

LIST OF TABLES

Table 6.1 Basic Specifications of 12 ISCAS 89 Sequential Benchmark Circuits 69

Table 6.2 Testing Time and Number of DFFs, Gates, Inverters, and Wires of Benchmark Sequential Circuits 70

Table 6.3 Fault Coverage, Number of DFFs, Wires and Detected Faults 71

Table 6.4 Number of Injected Faults, Test Vectors, and Detected Faults 72

Table 6.5 Simulation Results on ISCAS 89 Sequential Benchmark Circuits 73

LIST OF FIGURES

Figure 1.1 Digital logic circuit testing setup	**2**
Figure 1.2 A typical BIST environment	**6**
Figure 1.3 Conventional methodology of VLSI chip realization with automatic test generation	**6**
Figure 2.1 General process of event-driven simulation	**11**
Figure 2.2 Collapsing of stuck-faults	**14**
Figure 2.3 Fault lists in deductive simulation	**18**
Figure 2.4 Undetected faults versus number of test vectors	**20**
Figure 2.5 Reject ratio versus fault coverage	**23**
Figure 3.1 A NAND latch example	**26**
Figure 3.2 Time frame extension of NAND latch	**27**
Figure 3.3 General model for synchronous sequential circuit	**27**
Figure 3.4 Iterative combinational model	**28**
Figure 4.1 A D Flip-Flop	**37**
Figure 4.2 s27 gate-level schematic	**38**
Figure 4.3 Simulation of a D flip-flop under Altera MAX PLUS II	**40**
Figure 4.4 MAX PLUS II design environment	**41**
Figure 4.5 MAX PLUS II applications	**44**
Figure 5.1 The testing scheme in hardware at system level	**46**
Figure 5.2 Design flow of the sequential circuit testing method	**47**
Figure 5.3 An ALFSR structure	**48**
Figure 5.4 The fault injection scheme in hardware	**53**
Figure 6.1 Testing time vs. no. of wires and DFFs	**70**
Figure 6.2 Fault coverage vs. no. of DFFs and wires	**71**
Figure 6.3 Detected faults vs. injected faults, test vectors	**72**

LIST OF ACRONYMS

AHDL Altera Hardware Description Language
ALFSR Autonomous Linear Feedback Shift Register
ASCII American Standard Code for Information Interchange
ASIC Application Specific Integrated Circuit
ATE Automatic Test Equipment
ATG Automatic Test Generation
ATPG Automatic Test Pattern Generation/Generator
BIST Built-in Self-Test
BIT Built-in Test
CAD Computer-Aided/Assisted Design
CMOS Complementary MOS
CPU Central Processing Unit
DFF D Flip-Flop
DFT Design for Testability
EBT Extended BackTrace
HDL Hardware Description Language
IC Integrated Circuit
IEEE Institute of Electrical and Electronics Engineers, Inc.
IP Intellectual Property
I/O Input / Output
LAB Logic Array Block
LFSR Linear Feedback Shift Register
LPM Library of Parameterized Modules
LSI Large Scale Integration
MOS Metal-Oxide-Semiconductor
MUT Module Under Test
MUX Multiplexer
NS Nanosecond

PC Personal Computer

PCB Printed Circuit Board

PLA Programmable Logic Array

RAM Random-Access Memory

RCU Response Compaction Unit

ROM Read-Only Memory

RNG Random Signal Generator

S-A-0 Stuck-at-0

S-A-1 Stuck-at-1

SCF Simulator Channel File

SCIRTSS Sequential CIRcuit Test Search System

SDF Standard Delay Format

SNF Simulator Netlist File

SOC Systems-On-Chip

ST Self-Test

STG State Transition Graph

TPG Test Pattern Generation/Generator

TTL Transistor-Transistor Logic

ULSI Ultra Large Scale Integration

VHDL Very High Speed Hardware Description Language

VLSI Very Large Scale Integration

ABSTRACT

A Verilog HDL digital circuit fault simulator to detect permanent stuck-at logic faults for embedded cores-based synchronous sequential circuits is proposed in this thesis. The fault simulator can emulate a typical built-in self-testing (BIST) environment that utilizes a test pattern generator that sends its outputs to a module under test (MUT), with the resulting output from the MUT being fed into a test data analyzer. A fault is detected if the module response is different from that of the fault-free MUT. The fault simulator is suitable for testing synchronous sequential circuits described at the gate and flip-flop level in Verilog HDL. The subject thesis describes the detailed architecture and implementation of the fault simulator. Some simulation experiments on ISCAS 89 sequential benchmark circuits are also provided and discussed. The thesis also explores possible application of the ideas proposed to current embedded cores-based systems-on-chip (SOC) technologies, specifically in the context of testing memory-based synchronous digital systems.

Chapter 1
INTRODUCTION

Even though testing of digital circuits is an established area of research, it requires continuous revamping to keep pace with the rapid advances in integrated circuit (IC) technology. As the number of components on a chip and the gate to pin ration increase, the testing problem becomes more difficult, and the testing cost contributes to an increasingly larger proportion of the total product cost. Also, classical testing approaches would become obsolete, and novel and revolutionary approaches will be required to handle the ever-increasing complexity of Very Large Scale Integration (VLSI) devices.

The primary task of testing is to detect or discover the physical defects produced during manufacturing processes. In general, logic circuits are tested by a sequence of input stimuli, known as test vectors, which check for possible faults in the circuit by producing observable faulty responses at primary outputs. In this process, the most difficult part is the generation of a set of test vectors that will check faults at all points in the circuit, and therefore, uncover close to 100% of all possible defects in the chip.

1.1 Digital Circuits Testing – Basic Concepts

Testing is a critical part of the manufacturing process for a digital circuit, circuit board, or system. Testing a system is an experiment in which we exercise the system (for example, a chip) and analyze the resulting response to ascertain whether it behaved correctly or not. Figure 1.1 shows a digital logic circuit module test setup. The test vectors are produced by a test pattern generator (TPG) and applied to exercise the module under test (MUT). The operation of the MUT is evaluated by capturing its response to the test vectors and then comparing the produced response to the expected values.

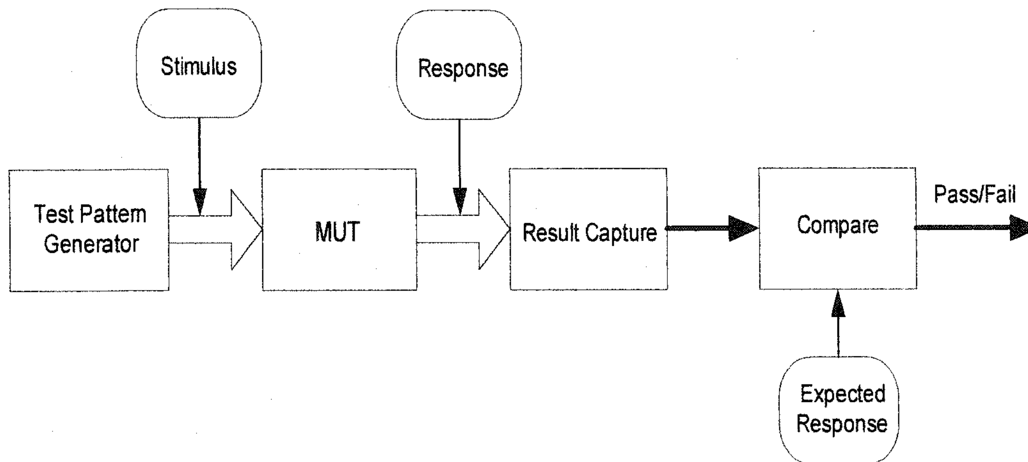


Figure 1.1. Digital logic circuit testing setup.

Testing can be carried out at different stages of the manufacturing process. The sooner a defect is detected, the lower is the cost. For instance, the approximate cost [5] to a company for detecting a fault at the various levels may be summarized as follows:

- wafer $0.01~$0.1
- packaged-chip $0.10~$1
- board $1~$10
- system $10~$100
- field $100~$1000

In general, testing could be of two basic types: functional testing and manufacturing testing. Functional tests are used to validate the correct operation of a system with respect to its functional specifications. They are usually implemented early in the design cycle to verify the functionality of the MUT. Functional tests are said to be exhaustive if they could detect almost any fault whatsoever. However, exhaustive testing be better used in small circuits because of the length of the resulting tests. Pseudoexhaustive tests could be significantly shorter than exhaustive tests and could be used if some prior information about the structure of the circuit is available, thereby narrowing the universe of search space.

Manufacturing tests try to verify if every transistor in the circuit operates as expected. They are used after the chip is manufactured to verify that the silicon is intact. Although the natural flow of design usually has a designer considering manufacturing concerns besides functionality, in most cases, these two kinds of tests might be similar.

The typical defects, which may be caused during the chip fabrication or accelerated life testing, are layer-to-layer shorts, discontinuous wires and thin-oxide shorts to substrate or well. These defects will lead to particular circuit maladies, including nodes shorted to power or ground, nodes shorted together and inputs floating, or outputs disconnected.

In order to determine the existence of the aforesaid faults in the circuits, a precise fault model is required. A fault model basically is the representation of the effect of a failure by means of the change that is produced in the system signals. The fault models help generate tests and evaluate test quality defined in terms of coverage of the modeled faults. The most popular model is the single stuck-at-fault model (i.e. stuck-at-one and stuck-at-zero). Other fault models include stuck-open or -close, bridging-fault or multiple-fault models. In most cases, the single stuck-at-fault model is extensively used because of its simplicity and accurate representation of a large class of logic faults.

When a designer or tester desires to measure the output of a gate within a large circuit to verify if it works correctly, and to assess the degree of difficulty of testing a particular signal within a circuit, observability and controllability become significantly important. The concepts of controllability and observability are defined as:

- Controllability: the ease of producing an arbitrary valid signal at the input of a component by exercising the primary inputs of the circuit.
- Observability: the ease of determining at the primary output of the circuit what happened at the output of a component.

Test evaluation is an important part of testing, and it determines the effectiveness or quality of a test. It is usually done in the context of a fault model. The quality of a test is usually measured by the fault coverage, which refers to the ratio between the number of faults detected and the total number of faults in a circuit. The test evaluation is carried out by fault simulation, a process of simulating the occurrence of various faults and determining if they are detectable by a given set of test vectors. If the response differs from the expected response of the fault-free circuit module, a fault is detected.

1.2. Testing Problem

Fault modeling: During chip fabrication, many types of defects can occur, for example, breaks in signal lines, lines shorted to ground, excessive delays, etc. In general, the effect of a fault is represented by means of a model. The most commonly used model is the single stuck-at fault model. In this model, one assumes that any one signal line in the circuit may have a fault such that this signal line is permanently held at either a *logic 1* or a *logic 0*, irrespective of the input vectors. The single stuck-at fault model has been found to be effective in representing the behavior of most faulty circuits, i.e., the test vectors that can detect a large number of single stuck-at faults can also detect most of the defects in the actual device. This model is simple to analyze because the faults are considered at the logic level that is independent of technology. Furthermore, the total number of modeled faults is at most twice the total number of signal lines in the circuit. This number can be further reduced by fault collapsing.

Test generation: The problem of generating a test for a given fault has been proved to be NP-complete even for combinational circuits [6] [7]. Although some test generation methods guarantee a test if one exists for a fault in a combinational circuit, the NP-completeness property necessitates the use of clever heuristics in actual practice. Test generation for purely sequential circuits poses additional complications due to the presence of memory states.

Measures of test quality: The quality of tests is measured in terms of the size (or length) of the test sequence and the fault coverage. The length of the test sequence determines the time required to test the actual device on automatic test equipment. The fault coverage is defined as the fraction of modeled faults detected by the test sequence. It is evaluated by fault simulation that computes the test response of the fault-free and the faulty circuits. From the simulation results, one can determine which faults are detected. It is desirable to achieve high fault coverage.

1.3. Test Generation Techniques

Large-scale integration has added enormous complexity to the process of testing. This complexity is due mainly to the reduction in the ratio of externally accessible points (primary inputs and outputs) to internal inaccessible points in the circuit. We can classify the

integrated circuit testing techniques into three broad categories, namely: (1) testing of purely combinational circuits or synchronous sequential circuits using scan type of design for testability (DFT) techniques; (2) self-testing circuits that generate their own test vectors using built-in hardware; and (3) testing of general digital (sequential) circuits with test vectors that are externally generated and applied.

For purely combinational circuits, a number of methods are known that automatically generate tests with satisfactory fault coverage. For synchronous sequential circuits, scan design is often used to reduce the test generation problem to a combinational circuit testing problem that is considerably less difficult. In scan design, all memory elements of the circuit are chained into one or more shift registers such that a synchronous sequential circuit can be partitioned into a purely combinational circuit and a shift register that can be tested separately. While this technique has been successfully used in commercial systems, the 10-20% logic added for testability has performance and cost penalties that are not always acceptable.

Built-in self-test (BIST) is a test technique in which a circuit (chip, board, or system) can test itself; in other words, the testing (test generation, test application and response verification) could be accomplished through built-in hardware. Actually, BIST is a combination of two concepts: built-in test (BIT) and self-test (ST). This technique is intended to solve the some of major testing problems, like time and volume, the test cost and diagnosis.

The automatic test generation (ATG) produces the test vectors for application to the MUT and the response data from an entire test sequence are compressed into a single value called a signature, which is then compared to the signature of a fault-free circuit, as shown in Figure 1.2. A fault is detected if the test signature is different from the good signature. A typical BIST scheme is composed of an automatic test pattern generator, the module under test and a response compaction unit.

The test data analyzer consists of a response compaction unit, storage for the good signature, and a comparison unit.

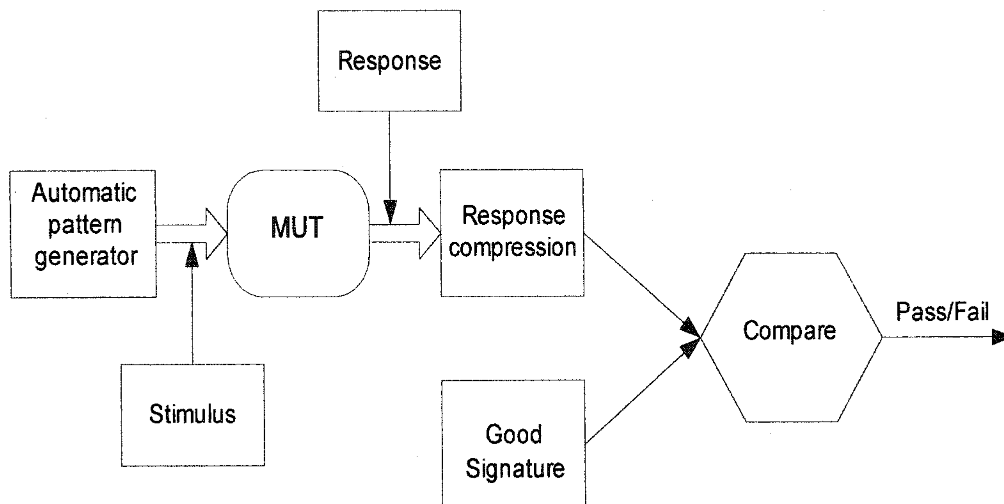


Figure 1.2. A typical BIST environment.

In BIST approach, the circuit module is designed to have a self-test mode in which it generates test vectors and analyzes the response. The objective is to apply all possible vectors to the combinational part of the circuit. In very large circuits, either the combinational portion is partitioned into independent sections or the whole circuit is tested by random test vectors. However, the hardware overhead of BIST is even higher than the scan design; 20% test logic may have to be added for BIST.

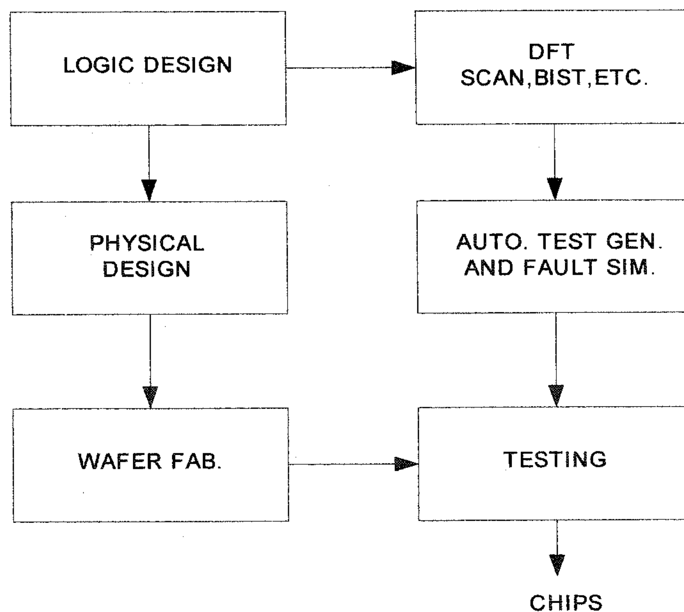


Figure 1.3. Conventional methodology of VLSI chip realization with automatic test generation.

Figure 1.3 shows the design for testability (DFT) methodology of VLSI chip realization. Because of the hardware overhead penalty, a majority of VLSI chips manufactured today contain neither scan nor built-in self-test capability. Most of these chips contain sequential logic. The test generation problem for general sequential circuits is recognized to be very difficult, tedious, and as yet, unsolved. The memory elements contribute to the poor controllability and observability of logic. Most test generators for sequential circuits can perform reasonably well only on circuits with up to 1,000 gates. Therefore, VLSI designers manually develop test vectors using the knowledge of the functional behavior of the circuit. These tests are generated to exercise critical paths and functions with selected data patterns. In spite of the enormous effort, the quality of manually generated test vectors, as often verified by fault simulation, appears questionable.

There thus exists a critical need to develop an automatic sequential circuit test generation methodology that can handle VLSI chips and, at a reasonable computing cost, achieve high fault coverage. This is the main objective of the present work. Equally important is the objective to combine the two processes of test generation and fault simulation.

1.4. Organization of the Thesis

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

Chapter 2 introduces the concepts of logic modeling, event-driven simulation, fault modeling, and fault simulation.

Chapter 3 describes several test generation approaches that are representatives of the important work in this area. Specific situations responsible for the high complexity and poor performance of these approaches are also discussed.

Chapter 4 gives an overview of Verilog HDL to describe the sequential circuits and simulate the sequential circuits under the Altera MAX PLUS II design environment.

Chapter 5 focuses on the detailed design and implementation of the new fault simulation method.

Chapter 6 presents test cases and experimental results, and Chapter 7 summarizes the work, provides a general conclusion, and describes future trends in this field.

Chapter 2
LOGIC SIMULATION AND FAULT ANALYSIS

Knowing the response of every component in a complex design does not imply that the entire system will function correctly. The system response is usually obtained through simulation. Since our objective in the thesis is to present a unified methodology for simulation and test, an understanding of the concepts described in this chapter is essential. More specifically, we introduce the concepts of logic modeling, event-driven simulation, fault modeling, and fault simulation.

2.1. Simulation

Simulation normally refers to obtaining the response of a system from a model. For electronic systems, both hardware and software models are used. While hardware models or breadboards are the traditional way of verifying designs, software models are becoming more popular due to their economy and accuracy, derived largely from advances in computing.

2.1.1. Circuit modeling

Digital circuits are modeled as interconnections of functional elements. The interconnections are described using a hardware description language. Generally, the level of modeling refers to the amount of functionality that is included in the elements. At the highest level, commonly known as the behavior-level, the elements are large functional blocks described in programming languages like C or Pascal. The next lower level is the gate-level which consists of Boolean gates like *AND, NAND, NOR, NOT,* and *OR.* Next is the MOS transistor level, also referred to as the switch-level. The lowest level is the circuit-level where components like transistors, resistors, capacitors, etc. are described through their electrical characteristics.

In order to effectively deal with the complexity of very large scale integration (VLSI), mixed-level models are usually used. In a large system, blocks of preverified designs may be

modeled at the behavior level. Other portions of the same system may be modeled at the gate or transistor levels.

Another useful concept in describing VLSI systems is hierarchy. In a digital system, functional blocks like registers, adders, and multiplexers may be described as interconnections of logic gates. The system then can be described as an interconnection of these functional blocks. If the system uses a particular functional block repeatedly, then the details of this block are described only once in the hierarchical description.

2.1.2. Signals

In a digital system, the structure of the circuit is not the only thing that is modeled. The signals flowing through interconnections must also be modeled. Real signals in an electronic system are voltages and currents. Strictly speaking, these are analog quantities. However, for digital systems, the common methodology is to model them as having discrete values. Most simulation systems use either three values (0, 1, X) or four values (0, 1, X, Z). The first two values 0 and 1 are the false and true logic states. The third value, X, is used to represent the unknown or ambiguous state of a signal and the fourth state, Z, represents the state of a floating node in the circuit. Based on the specific situation, a floating node may or may not retain its previous value.

Another essential attribute of signals is time. All signals are represented as functions of time. More appropriately, they can be considered as waveforms.

2.1.3. Modeling of delays

All circuit elements manipulate the signals supplied to their input port and produce the resulting signals at their output port. The manipulation of signals, however, takes finite time. Thus, irrespective of their speed, all electronic circuits involve delays. The delay is modeled in many different ways.

The simplest method is to assume that the elements (functional blocks, logic gates, or transistor switches) have zero delay. This assumption works well if the interconnections

involve no feedback paths. The zero-delay model can be effectively used to analyze combinational circuits that have no memory states.

When the circuit contains feedback paths or memory elements (e.g., flip-flops), it is necessary to maintain the order in which signals change. Under fairly general conditions, it is possible to maintain the proper sequence of events (signal changes) by a unit-delay model. In this model, each element is assumed to have one unit of delay. Since actual delays of elements are not all equal, the actual interval of time that this unit represents is meaningless. The unit-delay model is generally used for logic verification of gate-level and switch-level circuits. This model is also very popular for fault simulation discussed in a later section.

While the unit-delay model can verify the logical behavior of a digital circuit, it is inadequate for analyzing the timing behavior. For the timing behavior, a multiple-delay model is used. Each element is assigned a delay which is an integer multiple of a time unit. The time unit can be 1 nanosecond, 1 picosecond, or some such interval. Sometimes separate rise and fall delays are specified. It is also possible to specify propagation delays for interconnecting paths.

Most digital circuits can be analyzed by an appropriate discrete delay model as mentioned above. There can be situations, however, where a continuous delay model is necessary. Certain interconnections of bidirectional MOS devices (e.g., bus structures) and mixed analog-digital circuits are some examples requiring a fine-grain timing analysis. For VLSI circuits, mixed-mode analysis, where different components of the circuit are modeled at the appropriate level of timing accuracy, is perhaps the right way.

2.1.4. Simulator

A simulator is a computer program whose inputs are the circuit description and the primary input signal description or the stimuli. The stimuli can be specified as waveforms of the input signals. However, for digital circuits, a popular way is to specify them as vectors. A vector contains the values of all input signals. Whenever one or more inputs change, a new vector must be specified. In most circuits, input signal changes are synchronized with some periodically changing clock signal. Thus, the input stimuli are a sequence of vectors applied at specified periodic intervals.

The simulator computes the response of the circuit. Some simulators were compiled-code simulators. Entire circuit description was converted into the form of a computer program. Execution of this program with the stimuli as data would then lead to the circuit response. Since signal changes propagate from inputs to outputs, the circuit is levelized from inputs to outputs and the components are evaluated accordingly.

In a digital circuit, typically, at any instant only about 10% or fewer gates are active. Larger circuits tend to have smaller activity. However, the location of this activity changes with time. Circuits with components having different delays are also difficult to simulate. A compiled-code simulator takes no advantage of the low activity and evaluates all elements.

Accurate and efficient simulation of delays, feedbacks, and the dynamically changing activity is possible by the event-driven method described next.

2.1.5. Event-driven simulation

In the simulation terminology, signal changes are called the events. An event is characterized by the signal name, the type of change, and the time of change. A signal is assumed to retain its value until its source (e.g., the gate generating the signal) produces an event. A gate, on the other hand, cannot produce an event at its output, unless some event causes a change at its input. An event-driven simulator, therefore, simply follows the paths of events in the circuit. The simulator also deals with the problem of simultaneous events by analyzing all events occurring at a time before analyzing any events that would take place in the future.

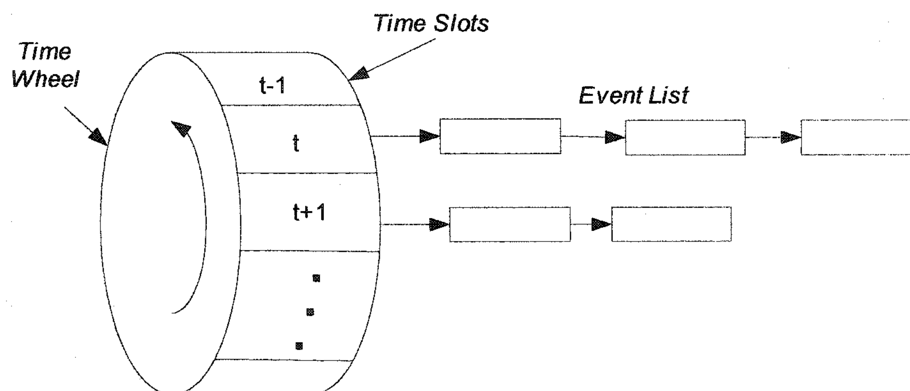


Figure 2.1. General process of event-driven simulation.

The general process of event-driven simulation can be easily understood from Figure 2.1. The time, over which the circuit activity is analyzed, is divided into some suitably selected unit. All delays in the circuit are specified in terms of this time unit. The simulator contains a circular stack known as the time wheel [8]. Each element of this stack, called the time slot, represents one unit of time. The number of time slots in the time wheel should not be less than the largest delay of any element in the circuit.

Each time slot contains a pointer to a list of events that are to take place at the corresponding time. Since the number of events at any time can vary, the events are usually stored in a linked list. Primary input changes also produce events on the input signals. As explained above, an event is described by specifying the signal name, the new value, and the time of change. The signal name and new value are stored in the event list, while the time of change is set by attaching the event to the appropriate time slot.

The simulator processes all events attached to the current time slot. Suppose, we denote the current time as t. Processing of an event means the following: (1) change the signal value, (2) evaluate all gates (or blocks) that now become potentially active due to an input signal change (this is done by tracing the fanout of the signal changed by the event), (3) if the output of any potentially active gate changes, then schedule the corresponding event. For scheduling an event, the simulator considers the delay of the gate whose output is changing. If the delay of that gate is d units, then the new event is attached to the event list of the time slot representing $t+d$.

Once an event is processed, i.e., all potentially active gates are evaluated and all new events are scheduled, the parent event is removed from its event list. When the event list of the time slot t becomes empty, the current time is advanced to $t+1$ and further event processing continues.

2.2. Fault Modeling

Just like any other analysis, fault analysis requires modeling (or abstraction). Fault models serve two purposes. First, they help generate tests, and, second, they help evaluate test quality defined in terms of coverage of modeled faults. A good fault model is one that is simple to analyze and yet closely represents the behavior of physical faults in the circuit.

Fault modeling is strongly related to circuit modeling. Consider a digital circuit described as an interconnection of logic gates. Even a moderate amount of imagination provides numerous fault possibilities, for example, missing gates, wrong gate types, missing interconnections, added interconnections, shorted interconnections, etc. Most of these faults, although physically real, are too complex to model. The faults most commonly modeled are stuck-faults.

2.2.1. Stuck-faults

Stuck-faults are not only the simplest faults to analyze, but they also have proved to be very effective in representing the faulty behavior of actual devices. The simplicity of stuck-faults is derived from their logical behavior; these faults are often referred to as logical faults. Stuck-faults was essentially proposed by Eldred [9] in 1959. One of the earliest theoretical analyses and discussions on stuck-faults was given by Poage [10].

Stuck-faults are assumed to affect only the interconnections. Possible fault sites are the inputs and outputs of gates. Each line can have two types of stuck-faults: stuck-at-1 and stuck-at-0. Thus, a line with a stuck-at-1 fault will always have a logical value 1 irrespective of the correct logical output of the gate driving it.

In general, several stuck-faults can be simultaneously present in a circuit. A circuit with n lines can have 3^n-1 possible stuck-line combinations. This is because each line can be in any one of the three states: stuck-at-1, stuck-at-0, or fault-free. All combinations except the one with all lines in the fault-free state are counted as faults. It is easy to realize that even with moderate values of n, the number of multiple stuck-faults will be very large; therefore, in practice, we only analyze single stuck-faults. An n-line circuit will have 2n single stuck-faults, a number that can be further reduced by fault collapsing.

2.2.2. Fault collapsing

Two faults are called equivalent if their effect is indistinguishable at the outputs of a circuit, which means that any test detecting one of them will also detect the other. Selecting one representative fault from each class of equivalent faults is called equivalence fault

collapsing. Computationally, this is an intractable problem in its general form. In practice, incomplete yet substantial fault collapsing may be possible with little computational effort. A good example is the collapsing of faults associated with the inputs and the output of a logic gate. Consider an n-input AND gate with none of the inputs directly observable. It is easy to see that an input stuck-at-0 (it is a common practice to write a stuck-at-i as an s-a-i) is equivalent to the output s-a-0; there is no way to distinguish between the two faults by observing the inputs and outputs of the circuit. For the purpose of test generation, therefore, we only consider n+2 of the 2n+2 faults associated with the AND gate: s-a-1 on each input and s-a-1 and s-a-0 on the output. Similarly, in an n-input OR gate, we test for s-a-0 on each input and s-a-1 and s-a-0 on the output. Testing considerations for NAND and NOR gates are similar.

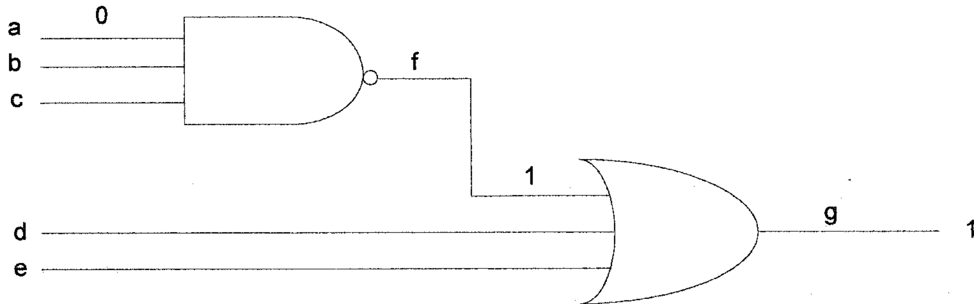


Figure 2.2. Collapsing of stuck-faults.

A three-value logic simulator is effective in finding equivalent faults that may be separated by several gates in a circuit. The three values used in simulation are 0, 1, and X= unknown or don't care. Consider the circuit of Figure 2.2. A value 0 on line a uniquely (i.e., irrespective of the other inputs of the NAND gate) forces a 1 on line f, and, thereby, forces a 1 on line g. This is easily checked by setting don't care values on all other input lines and noticing that the values of lines f and g are still unique. Thus, the faults, a stuck-at-0, f stuck-at-1, and g stuck-at-1, are equivalent.

The illustration above can be generalized. To find if a fault, say, an s-a-i can be collapsed with another fault, we carry out three-value simulation after initializing line a to i and all other lines to X. If, as a result of simulation, another line, say, b, is forced to a binary value j, then the fault an s-a-i is equivalent to b s-a-j. This simple procedure has been used in a MOS simulator where the circuit is described as an interconnection of transistors [11].

In equivalence fault collapsing, we only collapse the faults that are indistinguishable. If we are prepared to give up on diagnostic resolution (ability to distinguish between faults), more collapsing is possible. This is accomplished by using the concept of fault dominance as explained next. In very large scale integration (VLSI) circuits, where coverage of faults rather than their exact location is the overriding consideration, dominance fault collapsing may be desirable.

Consider two faults f1 and f2. Suppose all tests for f1 also detect f2 but only some of the tests for f2 detect f1. Then f2 is said to dominate f1. This definition of dominance is originally given by Poage [10]. However, some people have used the term dominance in an opposite sense. If we had to pick one to detect, obviously, we are safer to take f1. Even though, at times, it may be a little harder to find a test for f1, this test is guaranteed to cover f2.

In an AND gate, the output s-a-1 dominates any input s-a-1. If we desire dominance fault collapsing, then for an n-input AND gate we need to consider only n+1 faults.

Equivalence and dominance fault collapsing may be used to reduce the number of faults that must be considered for detection. In the AND gate example above, we found that the two stuck-faults on the output line can be collapsed with appropriate input-line faults. This type of collapsing can be repeatedly used until one arrives at a checkpoint defined as either a primary input or a fan-out branch. It has been shown that it is sufficient to consider single faults on checkpoints in a circuit as long as all such faults are detectable. In actual circuits, however, there can be a small number of faults, referred to as undetectable or redundant faults, which are not detected by any test. The presence of such faults, by definition, does not cause malfunction in the circuit. If any checkpoint fault is undetectable, then additional faults must be considered. Finding checkpoints in sequential circuits requires further analysis.

In sequential circuits, fault collapsing is often accomplished through multiple passes. In other related work, fault set reduction algorithms have been studied [12] [13].

2.2.3. Other fault models

Most VLSI designers encounter situations where stuck-type fault model may not be suitable. Two common cases are the memory and programmable logic array (PLA) blocks. For memories, stuck-fault model is still used for I/O registers and address decoders. However, for the storage cell array, the faults generally modeled are: (1) single cell stuck-at-1 or -0, (2) adjacent cell coupling, and (3) pattern-sensitive faults. Like memories, PLA fault models are also closely related to their structure. These include cross-point faults and bridging faults. For the purpose of logic and fault simulation, a PLA may be modeled as its two-level logic implementation. Even though stuck-type faults in this implementation are not a true representation of PLA faults, they are easy to analyze and are widely used. Cross-point and bridging faults can be represented in the two-level logic model by adding extra gates [14].

While the above described fault models are used in today's design environment, there are others that are gaining importance due to the changing technology. Some of these are the CMOS stuck-open faults, functional faults in microprocessors, and delay faults.

2.3. Fault Simulation

Simply stated, a perfect chip test must meet two criteria: reject every bad chip and pass every good one. Since the second requirement is not difficult to fulfill, we will focus on how well the first criterion can be met. Suppose a total of N chips passed the test and there are M bad chips among them. Then the fraction M/N, sometimes called the reject ratio of the tested product, is a measure of deficiency of the test. Unfortunately, appealing as it may be as an indicator of test quality, accurate reject-ratio data are hard to obtain in practice. Instead, an indirect but easier-to-estimate measure is used. It is called fault coverage and is defined as the percentage of modeled faults detected by the test. A good (or acceptable) test may be defined as one that achieves a certain minimum percentage of fault coverage (typically, in the high 90s). Accurate evaluation of fault coverage requires the use of a fault simulator, often working with the model of the circuit at just one level (e.g., transistor), switch, or logic gate. However, with increasing circuit densities, complete fault simulation using a low-level nonhierarchical circuit description is becoming very expensive and time consuming. Hierarchical simulators are being developed in response to this need.

2.3.1. Methods of fault simulation

Given a set of faults and a set of input vectors, a fault simulator must find out which faults are detected by the input vectors. Production-quality fault simulators at the logic-gate level were available well before the advent of large-scale integration (LSI) circuits. Recently, fault simulators at the transistor level have been developed for more accurate modeling of faults.

The simplest method of simulating faults is the serial fault simulation. A single fault is introduced in the circuit and simulation is run like the true-value simulation. Response is compared with the stored response of the fault-free circuit. As soon as the fault is detected, the current simulation is suspended and a new simulation is started with another fault. While this method has found some success in the hardware accelerator based simulators, in most commercial simulators, an effort is made to reduce computation time by simulating more than one fault in one pass for a given input vector. The simplest such technique is parallel fault simulation [15] in which a computer word of W bits is associated with each line in the circuit. During a pass through the simulator, each bit at a particular position would be associated with the circuit that has a specific single fault (or no fault). After simulation, the bit would store the value on the line in the associated circuit. Before the beginning of a pass, a set of (W-1) as-yet-unsimulated faults is chosen, where W is the word size of the computer. The remaining bit is used to simulate the fault-free circuit. The gates whose input/outputs are directly affected by a selected fault are flagged. When a flagged gate is simulated, the effect of the fault is injected into appropriate bits of the words representing its inputs and outputs. Each fault simulation passes starts at the primary input lines and proceeds in a breadth-first fashion towards the primary outputs. For each gate, fault simulation amounts to computing the output word as the gate's logical function of its input words. At the end of the pass, the word at each primary output can be examined to determine which of the simulated faults would be detected at that output. This is done by comparing the response of each faulty circuit against the fault-free circuit. As logical operations on computers can be carried out at the word level, W-1 different faults are simulated in parallel in each pass; hence, the name of the method. A total of F faults would be simulated in F/ (W-1) passes.

The ultimate, in terms of reducing the number of passes, is the deductive fault simulator [16]. It needs just one pass for simulation independent of the number of faults simulated. The basic idea here is to associate with each line a list of just those faults sensitized to that line (that is, signals on the line for the normal and faulty circuits are different) by the simulated input vector. Simulation of a gate requires deducing the fault list at the gate output from the input fault lists. This is illustrated for a 2-input AND gate shown in the example of Figure 2.3.

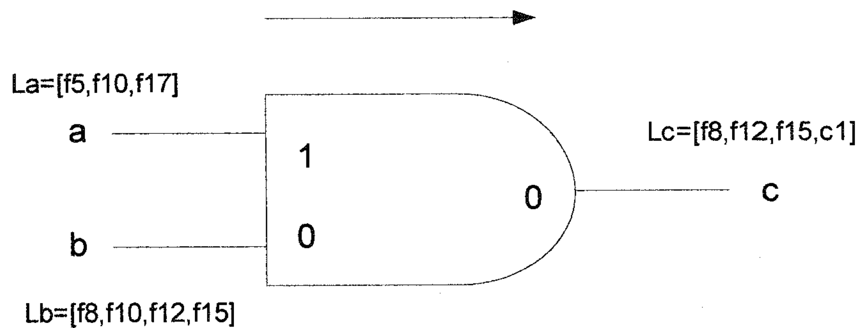


Figure 2.3. Fault lists in deductive simulation.

The gate is assumed to be embedded in a larger circuit. The fault lists associated with the inputs a and b of the gate are as shown; these would have been computed already in previous steps of the algorithm. The signal values shown are for the fault-free circuit. Consider the effect of the fault f5, appearing in La but not in Lb, on the gate output. The signal value on line a will change from 1 to 0 while that on line b will remain unchanged. Thus, the output will stay at 0; so f5 cannot be in the fault list Lc. Indeed, no fault in La can occur in Lc, since the output would not change under the fault. Next, consider the fault f8, which is in Lb but not in La. Such a fault will change the values on lines b and c and hence, must be included in Lc. Additionally, the fault "c stuck-at-1" (or c1) must be in the output fault list since it complements the normal value. Thus we obtain the expression $Lc= (Lb \cap \overline{La}) \cup [c1]$ for the output fault list shown in the Figure 2.3.

Compared with parallel fault simulation, the penalties paid for a single-pass in deductive simulation are: (1) dynamically varying storage for fault lists associated with each line and (2) more complex processing of gates requiring set operations on the line fault lists.

We note that the output fault list computations are essentially dynamic. The expression for the output list is not just a function of the gate type; it also depends on the signal values

on the gate inputs. Interestingly, the output fault list may change even though the gate inputs and outputs remain unchanged. This is because a line's fault list may change even when its value does not. For example, when the normal value on line a changes to 0, the value on line c is not changed, but Lc may change nevertheless. This "fault-list event" must be propagated through all gates to which line c is an input. This undesirable characteristic of the deductive method is absent from concurrent simulation to which we shall turn next.

The basic idea behind concurrent fault simulation [17] is quite simple. Typically, a fault changes very few signal values in a large circuit. Thus, most, if not all, of the information for simulating a fault is contained in the "good-circuit" simulation. In concurrent simulation, the good circuit is simulated in its entirety, but a faulty one is simulated only for gates whose states differ from their good-circuit states. For logic gates, the state is simply the combination of input and output values; however, the concurrent method is general enough to handle arbitrary elements with stored states.

The term concurrent is derived from the fact that each faulty gate carries enough information for independent simulation of the associated fault. This is considerably more information than just the fault index used in deductive simulation. Thus the speed is gained in concurrent simulation at the expense of additional dynamic storage per node in the circuit. There is more to the concurrent method, however, than indicated by this simple example. The ease with which it can be adapted to any level of simulation is evident in the following sample of implementations: FMOSSIM [18] at the switch level, MOTIS [11] at a mixed level, and the CHIEFS hierarchical fault simulator [20]. The ability of concurrent simulation to evaluate each faulty gate independently makes it the method of choice for implementations on hardware accelerators [21].

2.3.2. A comparison of fault simulation methods

The process of fault simulation can be likened to that of shooting at a progressively diminishing target. Initially, even a randomly chosen test vector is likely to detect many undetected faults, but as their number decreases, this strategy produces diminishing results. Figure 2.4 shows a typical curve of undetected faults versus number of tests. It can be partitioned into two phases where random test generation is seen to be a productive strategy

for Phase I (the exponential decay part of the curve). There is a gradual transition from Phase I to an almost linear decay in Phase II.

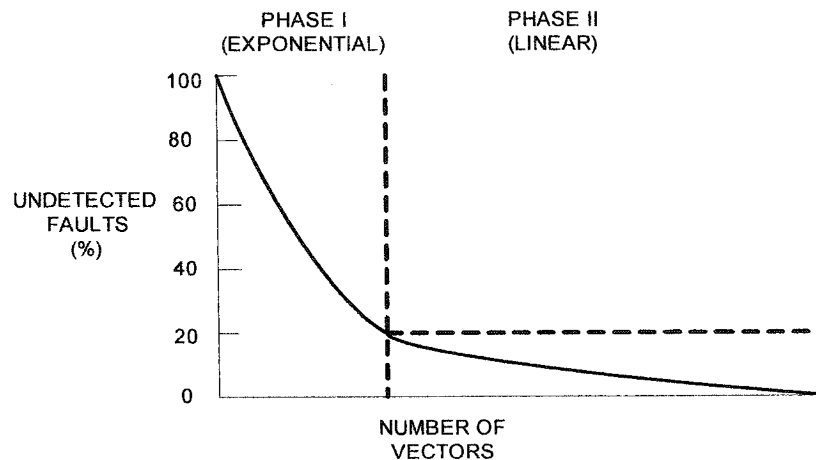


Figure 2.4. Undetected faults versus number of test vectors.

Empirical evidence suggests the transition point to lie somewhere between 35 and 15 percent of undetected faults. Taking such a curve as the model, it has been estimated [22] that the simulation costs for a network with G gates grow proportional to G^3 and G^2 for parallel and deductive (or concurrent) simulations, respectively. A concurrent fault simulator at the transistor level is reported to take 1527 seconds (approx. 25 minutes) of computation time for simulating a 9,478-transistor circuit on a 12-MIP computer [11]. A set of 2,241 vectors was used in this simulation. According to the quadratic rise in simulating time, a 100,000-transistor circuit would require over 47 hours on the same machine. Clearly, brute-force simulation of large circuits is not a practical proposition even for the currently achievable circuit densities. Attractive alternatives are to use statistical sampling techniques or hardware accelerators [24].

Another way to compare fault simulation methods is to analyze their capabilities. One such comparison was carried out for the parallel, deductive, and concurrent methods [25]. The metrics of comparison were abilities to handle multiple signal values, different levels of abstraction, and accurate timing. The concurrent method scored the highest on all three measures. The parallel method came a distant second with the deductive not too far behind it.

2.4. Fault Simulation Tools

VLSI CAD systems normally incorporate a fault simulator. These fault simulators have three parts: (1) preprocessing, (2) fault simulation, and (3) output analysis.

A fault simulator processes the circuit connectivity description in much the same way as a true-value simulator. Often, true-value simulation for design verification precedes fault simulation, and a preprocessed or compiled description of the circuit already exists. Additionally, the fault simulator makes a fault list of all stuck-faults on the inputs and outputs of gates and the functional blocks. Some fault simulators provide an option to include transistor faults. If the circuit contains functional blocks (e.g., memories or other blocks described in some high-level language), then stuck-faults are modeled on the inputs and outputs of those blocks. The fault list is processed to collapse equivalent faults. The user can select or remove any faults from the fault list or request a random sample of given size. User inputs also include specification of the signals to be monitored (e.g., all primary outputs) and the strobe positions. Many fault simulators allow user selectable options for handling of race and oscillation faults. These faults may introduce potentially unstable states in the circuit and are often represented by the unknown value (X) in simulators. If an unknown state due to a fault propagates to a visible point, the fault can be considered as potentially detectable.

The second part of a simulator performs its main function: simulation of the circuit with selected faults by using the given vectors. It keeps the detection data (time and output of detection) for each fault and also stores the true-value response. Most simulators drop a detected fault from consideration once it is detected. Race and oscillation faults may also be dropped at user option.

Since fault simulators use a large amount of computing resources, the normal practice is to divide the vector set according to the available memory in the automatic test equipment (ATE). Each vector set (often called a vector load) is run through the simulator by using the list of undetected faults up to that point. At the end of a run, the simulator can store the internal states of the circuit in a checkpoint data set for use by the next vector set. However, if the ATE requires each vector set to reinitialize the circuit, the simulator checkpointing will

give a coverage that is too optimistic. Also, for realistic fault detection data, the user should set the observation strobes in the same way as the ATE's strobes.

The results of the second part are fault coverage and the expected response. The latter is provided to the test program. The third part of a fault simulator analyzes the fault coverage data and displays it in an easy-to-interpret form. A popular way of presentation is a graph showing fault coverage as a function of the vector number in the test sequence. The simplest way, of course, is to list faults under the following categories: detected, undetected, race, and oscillation. These lists may be used by other programs such as automatic test generators. Displaying undetected faults on the schematic can be a great help in manual test generation. For an undetected fault, tracing the activity caused by this fault can also be useful to the designer in writing a test.

Another effective method of displaying the fault simulator result would be to display the detected faults on the chip layout. Such a display will easily isolate the portions of the chip where fault coverage needs enhancement.

2.5. Fault Coverage and Product Quality

Independent of whether the whole class of single line stuck-faults is simulated or only a sample thereof, the obtained fault coverage is at best an imperfect measure of test effectiveness. This must be so because a fault simulator cannot evaluate the coverage of actual physical faults (shorts or opens in metal, diffusion, or polysilicon, shorts between layers, parametric irregularities, etc.). Multiple faults are also not simulated because of their very large number, even though with the small geometries used, a processing defect is quite likely to lead to multiple faults. The obtained value of fault coverage may also depend on the specific simulator used for evaluation, since different simulators could employ different criteria to detect fault-induced races and oscillations. Redundant lines are not uncommon in real circuits though they may not always be identifiable because of the provable intractability of the problem. The effect of unidentified redundancies is unduly pessimistic fault coverage.

In spite of the above reservations, fault coverage data provided by simulators continue to be relied on as a figure of merit for a test. One may consider that how this figure of merit is related to the quality of the tested chips? A quantitative answer to this question can be given

based on a model of fault distribution on a chip. In one such model, it is assumed that a random number of logical faults are spawned by each physical defect on the chip. The physical defects themselves are randomly distributed over the chip but form clusters for a variety of processing related reasons. Thus a compound distribution is used for logical faults [26] [27] with an appropriate clustered distribution used for physical defects. It is possible to estimate the parameters of the compound distribution from the wafer-level test data and produce the curve shown in Figure 2.5. The vertical ordinate in the figure is reject ratio. The horizontal axis represents the fault coverage for single stuck-type faults.

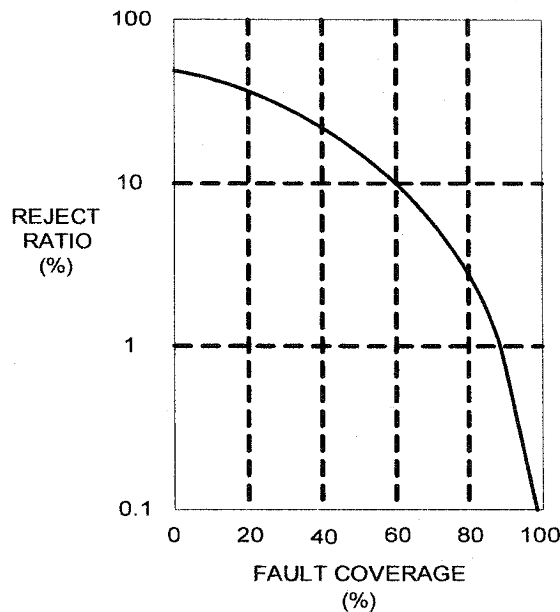


Figure 2.5. Reject ratio versus fault coverage.

Interestingly, it is observed that making a transition to a finer feature size, while keeping other parameters fixed, has the effect of moving the curve to the left. This means that for denser chips actually a lower value of fault coverage would suffice for a given value of reject ratio; an encouraging result considering the disproportionately high costs involved in increasing the fault coverage in Phase II of Figure 2.4 to cover such faults.

In some related works, a uniformly random defect distribution model is used for logical faults. The term is used as defect level to denote what we have called as reject ratio. Although the results have been applied to both chips and boards, the uniformly random fault distribution gives pessimistic results for chips where the defects may be clustered [28].

Chapter 3

TEST GENERATION APPROACHES

Test generation approaches can be classified into three categories: exhaustive, random and algorithmic. For combinational circuits, if the number of primary inputs is small, using exhaustive tests consisting of all possible input vectors to ensure 100% fault coverage is an obvious candidate. Exhaustive techniques can be extended to larger circuits by partitioning into subcircuits such that each subcircuit is tested exhaustively by a reasonably small number of test vectors. However, finding suitable partitions is neither easy nor guaranteed.

Random test generation is a simple and low-cost method in which input vectors are generated randomly. Outputs of the faulty and the fault-free circuits for each random vector are compared by a fault simulator. If any fault is detected, the random vector is retained as a test. The number of random vectors needed for high fault coverage can be very large in some cases. If the numbers of levels of logic and gate fan-ins are large, this approach becomes less effective. Random test generation techniques can be improved by generating vectors with selected input signal probability. Also, the random test generation methods can be augmented by deterministic methods.

Since early 1960s, numerous algorithms have been proposed for generating test vectors for combinational and sequential circuits. Some of these are effective and widely used in practice; others are of limited practical interest. Most approaches are topological, i.e., they construct the input vector by analyzing the circuit topology. Several famous algorithms for combinational circuits like D-Algorithm, PODEM and FAN are widely used and can usually produce satisfactory results. These are commonly known as path sensitization algorithms. Some algebraic methods have also been reported and are quite elegant and complete. However, their high complexity makes them impractical for large circuits.

Even though several sequential circuit test generators have been developed, their performance is questionable. They have one or more of the following limitations: (1) circuit must be synchronous, (2) circuit must have limited number of flip-flops, (3) must have limited number of gates, (4) must have limited number of vectors per fault, and (5) the circuit delays must be neglected.

In the following sections of this chapter, we review the basic concepts of the widely used path sensitization approach. Some representative algorithms for both combinational and sequential circuits are also discussed.

3.1. Test Generation for Sequential Circuits

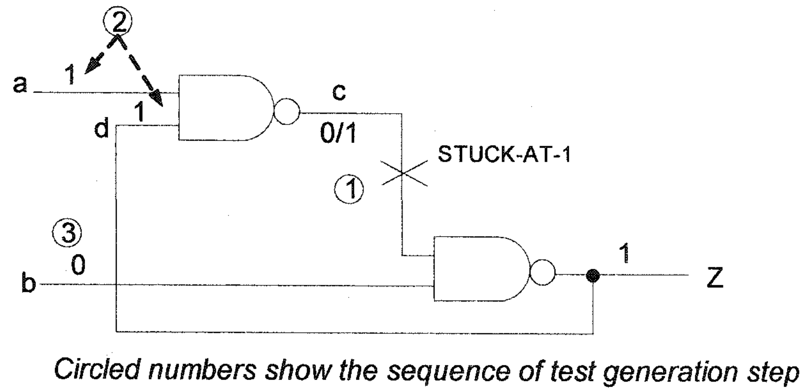
The response of a sequential circuit depends on its primary inputs and the stored internal states. The stored states can retain their values over time. Thus, combinational test generation methods can be applied to sequential circuits if the element of time is introduced.

3.2. Approaches for Test Generation

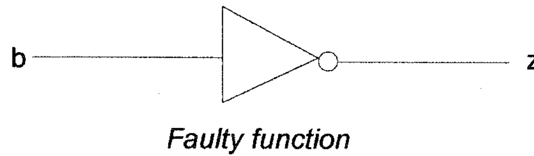
3.2.1. Iterative array approach

Many sequential circuit test generators have been devised on the basis of the fundamental combinational algorithms. A combinational model for a sequential circuit is constructed by regenerating the feedback signals from previous-time copied of the circuit. Thus, the timing behavior of the circuit is approximated by combinational levels. Topological analysis algorithms that activate faults and sensitize paths through these multiple copies of the combinational circuit are used to generate tests.

We illustrate this approach by a simple example. Consider the latch in Figure 3.1. Let us assume that all gates have zero delay. The result of applying the combinational test generation approach is shown in the figure. The process stops with inputs 1 0 and the output 1. Since the output is the same in both good and faulty circuits, a test is not found.



Circled numbers show the sequence of test generation step



Faulty function

Figure 3.1. A NAND latch example.

Figure 3.1 also shows that the faulty function, when "line c is stuck-at-1", is z $=\bar{b}$. The fault blocks the feedback and the circuit no longer has the storing capability. It can be easily detected by first applying a 1 0 input and then following it up by 11. The first pattern will produce a 1 output irrespective of the fault. The second pattern simply stores the state of the latch. In the good circuit, the output will remain 1 while it will change to 0 in the faulty circuit.

Our combinational test generator was not able to solve this problem due to the zero-delay assumption. If we consider finite delays of gates, it is possible to first apply the value of z = 1 to d and then change b to 1 to sensitize the path for the fault. A common practice is to cut the feedback path. This is shown in Figure 3.2. A copy of the circuit is attached to generate the feedback signal d. The test generation begins from the copy shown as current time frame. In general, any number of time frames (previous or future) can be added on either side of the current time frame. However, the complexity of the model increases with the number of time frames. Another problem occurs due to the signals left unspecified by the test generator. For example, the test in Figure 3.2 is a 1 1 pattern that is preceded by X 0 where X denotes the don't-care state. If we set X to 1, we get the desired test. But if X is set to 0, the test will cause a race in the fault-free circuit. Thus, sequential circuit tests generated by such procedures require special processing to avoid timing problems.

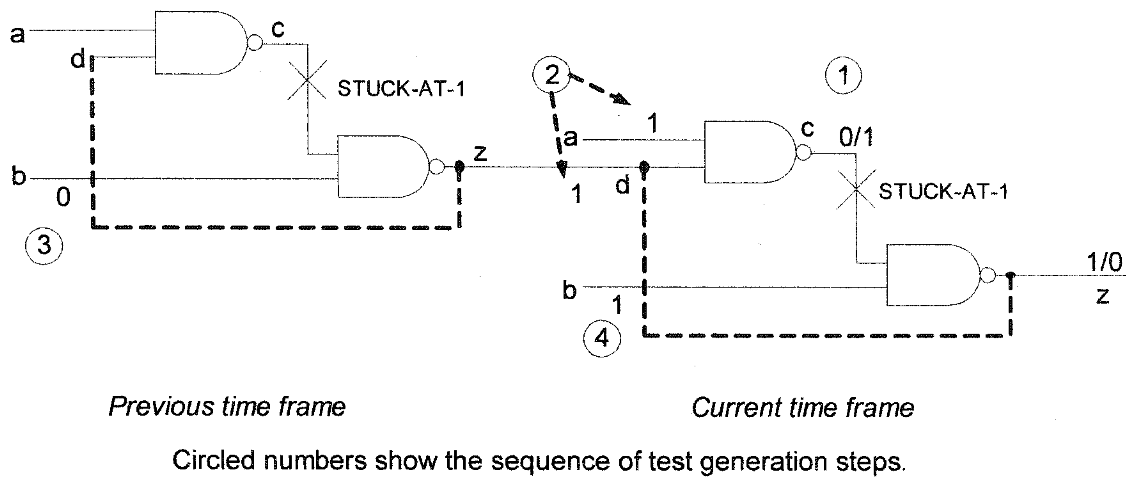


Figure 3.2. Time frame extension of NAND latch.

3.2.1.1. Extended D-algorithm

A general model for synchronous sequential circuit is shown in Figure 3.3.

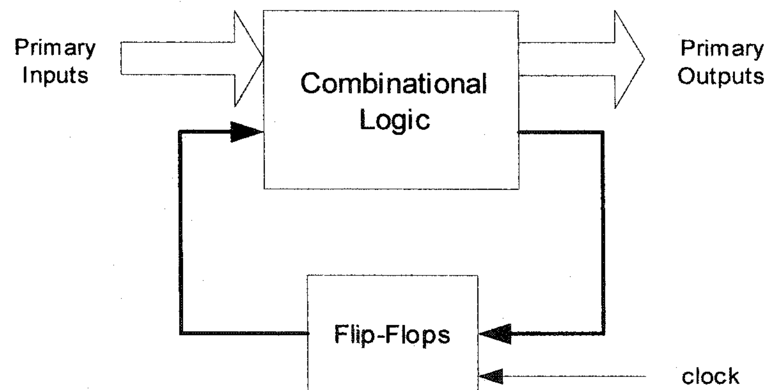


Figure 3.3. General model for synchronous sequential circuit.

The output is a logic function of primary inputs and the present state of flip-flops. This circuit can be modeled by a combinational network as shown in Figure 3.4.

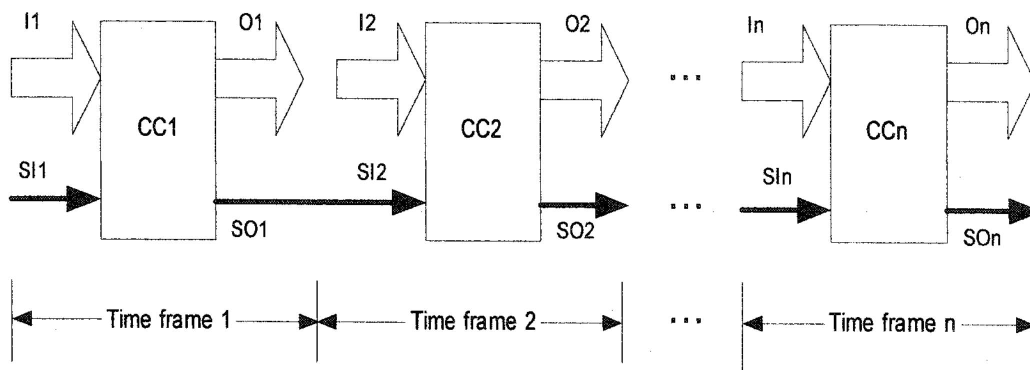


Figure 3.4. Iterative combinational model.

Feedback lines are cut and the combinational portion is duplicated. The block CCn is the nth copy of the combinational portion and corresponds to the time frame n. The inputs of CCn include the primary inputs In and the pseudoinputs SIn obtained from the feedback signals produced by CC_{n-1}. Similarly, On and SOn are the primary outputs and pseudooutputs of the time frame n.

In order to find a test sequence to detect a single stuck-at fault, we must first determine the number of time frames needed in the combinational model. The D-Algorithm [29] is then applied to the jth copy of the combinational model, called the current time frame. If the D-Algorithm assigns a value to any pseudoinput, this assignment is then justified in the (j-1)th copy. Similarly, if the fault effect is propagated to a pseudo-output, propagation must continue into the (j+1)th and subsequent copies until the fault effect reaches a primary output. It is worth noting that a single stuck-at fault in the sequential circuit corresponds to a multiple fault in the combinational model, since the complete model contains one fault per time frame. Therefore, the D-Algorithm applied to the combinational model should be modified to deal with fault multiplicity. The multiple fault consideration and the extra complexity of the replicated logic make this approach unrealistic for large circuits.

3.2.1.2. Nine-value algorithm

Muth [30] extended the five-value algebra used in the D-Algorithm to nine-value algebra. These nine values (1/1, 1/0, 1/X, 0/1, 0/0, 0/X, X/1, X/0, and X/X) represent ordered pairs of states of the fault-free and faulty circuits. The five values used in the D-Algorithm

are actually a subset of these nine values; a 1 used in D-Algorithm is equivalent to 1/1, a 0 is 0/0, an X is X/X, a D is 1/0, and a $\overline{D}$ is 0/1. The additional partly specified values, 0/X, 1/X, X/1 and X/0, provide a greater degree of freedom in test generation. The nine-value algorithm takes into account the possible repeated effects of the fault in a sequential circuit and a test can be generated for asynchronous circuits also. Even for synchronous circuits, the implementation of the algorithm is very complex and may be inefficient for large circuits.

3.2.1.3. SOFTG

Primarily, SOFTG [32] is a sequential version of PODEM [31]. Tests are constructed by tracing backward through the circuit in much the same way as a combinational test generator, but all forward signal propagation is carried out by an event-driven simulator. The close interaction with the simulator allows SOFTG to effectively model the timing behavior of a sequential circuit such that the tests will not cause races or hazards in the fault-free circuit. The method is very effective for circuits of moderate complexity (few hundred gates) but the backtracking for alternative choices in the combinational algorithm restricts its use for very large circuits. A similar idea is reported by Kjelkerud and Thessen [33]. They implemented the D-Algoritym by using a table-driven logic simulator for forward implication and a deductive fault simulator for D-propagation. The fault simulator can ensure that the vectors are hazard-free.

3.2.1.4. Backtrace algorithms

All sequential test generation algorithms discussed above have a process flow that is bidirectional in time where time refers to the time of events in the circuit. This is because the starting event is the fault activation at the site of the fault. In general, the detection will take place at a future time while the primary inputs must assume their values in the past. Marlett [34] proposed a unidirectional single path sensitization algorithm that works backward, both spatially and in time, from a selected output toward the fault. For a given fault, a path and an output pin with best observability are first selected. The test generator begins by setting up sensitizing conditions at the path output and proceeds backward. The unidirectional

(backward) time flow simplifies implementation. However, it might lose some accuracy because only one path is allowed to be sensitized at a time. This algorithm is generally known as the extended backtrace (EBT) and its implementations have produced practical test generation programs.

While the EBT algorithm sensitizes a preselected path, the other proposed BACK algorithm [35] only preselects a primary output to which it sensitizes all paths starting at the fault site.

3.2.2. Verification-based approaches

The verification approach relies on determining whether or not the module under test is operating in accordance with its state table. This requires finding a sequence that forces the circuit to go through all states and output transitions. Actually, this approach can be considered a form of functional testing. Unfortunately, the high complexity of building the complete state table limits its application for large circuits. In addition, test sequences are usually excessively long for highly sequential circuits. However, the technique has been successfully applied to protocol testing.

3.2.2.1. SCIRTSS

SCIRTSS (Sequential CIRcuit Test Search System) [41] divides the test generation process into three steps. The D-Algorithm is first applied to the combinational portion of the circuit. The test vector, thus obtained, is then split between primary inputs and the present state. At this point, the problem becomes one of finding an input sequence to bring the circuit to this state and finding another sequence to propagate the stored fault effect to a primary output. These two steps require finding suitable paths through the state graph. In order to limit the effort, the searches for these paths are only conducted over a restricted state diagram and a small portion of the much larger set of data states. The data states included in search depend on user-supplied heuristic information.

A test generator for finite state machines [36] applies PODEM with iterative-array type of processing for forward propagation. The backward justification phase is replaced by a

procedure that attempts to search for a path in the state transition graph from a given reset state to the present state required by PODEM. In order to limit the complexity of the state transition diagram, a partial state diagram is constructed containing all valid states of the finite state machine but only a few transition edges. In searching for a path through the state transition graph (STG), the STGs of faulty circuits are assumed to be the same as that of the fault-free circuit. Since this assumption may not be universally true, the generated vectors must be verified by a fault simulator.

3.2.3. Functional and expert system approaches

A functional approach [19] [23] uses a high-level description of the circuit to generate test sequences that will verify whether the designed functions are being performed correctly. This approach often uses restricted fault models like the line stuck-faults on inputs and outputs of functional blocks or higher-level fault models such as errors in the truth table of a combinational block, or a change in the state table of a sequential functional block. The functional testing is often the method of choice for testing very large circuits like microprocessors. However, it is difficult to evaluate the quality of functional test vectors. In many cases, these tests may not be capable of detecting every possible failure that can occur.

The expert-system approaches [42] [65] incorporate the knowledge of human test programmers and a variety of algorithmic and heuristic techniques into an interactive software environment. These approaches require intensive user interaction, higher-level modeling libraries, and often restricted architectures. For these reasons, it has been difficult to effectively integrate them into existing design methodologies.

Chapter 4

FAULT SIMULATION UNDER ALTERA MAX PLUS II DESIGN ENVIRONMENT

4.1. Designing Sequential Circuit with Verilog HDL

4.1.1. Developing Verilog hardware description language

Hardware description languages, or HDLs are the languages used to design hardware with. As the very name implies, an HDL can also be used to describe the functionality of hardware as well as its implementation. HDLs allow designers to describe designs at higher levels of abstraction, such as architectural or behavioral, and provide a path to logic synthesis.

HDLs allow the design to be simulated earlier in the design cycle in order to correct errors or experiment with different architectures. Designs described in HDL are technology-independent, easy to design and debug, and are usually more readable than schematics, particularly for large circuits.

The principal feature of a hardware description language is that it contains the capability to describe the function of a piece of hardware independently of the implementation. The great advance with modern HDLs was the recognition that a single language could be used to describe the function of the design and also to describe the implementation. This allows the entire design process to take place in a single language, and thus a single representation of the design.

Verilog HDL is one of the most common hardware description languages (HDLs) used by integrated circuit (IC) designers.

Verilog HDL also allows for mixed-level designs, where users can describe a design at both high and low levels of logic simulation and synthesis. Designers are choosing top-down design and mixed-level design to contend with ever-increasing complexities and shrinking time-to-market cycles.

The Verilog hardware description language, usually just called Verilog, was designed and first implemented by Moorby [37] at Gateway Design Automation. It was first used beginning in 1985 and was extended substantially through 1987.

The implementation was the Verilog simulator sold by Gateway. The first major extension was Verilog-XL, which added a few features and implemented the "XL algorithm" which was a very efficient method for doing gate-level simulation. This marked the beginning of Verilog's growth period. Many leading-edge electronic designers began using Verilog at this time because it was fast at gate level simulation, and had the capabilities to model at higher levels of abstraction. Users began to do full system simulation of their designs, where the actual logic being designed was represented by a netlist and other parts of the system were modeled behaviorally. In 1995, Verilog became an IEEE standard.

Many companies began working on Verilog simulators. Now, Verilog simulators are available for most computers at a variety of prices, and which have a variety of performance characteristics and features. Verilog is more heavily used than ever, and it is growing faster than any other hardware description language. It has truly become the standard hardware description language.

4.1.2. Digital circuit design and Verilog HDL

The Verilog hardware description language is widely used in both industry and academia for describing digital systems. The language supports the early conceptual stages of design with its behavioral level of abstraction, and the later implementation stages with its structural level of abstraction. The language provides hierarchical constructs, allowing the designer to control the complexity of a description. Now, the language is openly available for any tool to read and write.

Digital systems are highly complex. At their most detailed level, they may consist of millions of elements, as would be the case if we viewed a system as a collection of logic gates or pass transistors. From a more abstract viewpoint, these elements may be grouped into a handful of functional components such as cache memories, floating-point units, signal processors, or real-time controllers. Hardware description languages have evolved to aid in

the design of systems with this large number of elements and wide range of electronic and logical abstractions.

The creative process of digital system design begins with a conceptual idea of a logical system to be built, a set of constraints that the final implementation must meet, and a set of primitive components from which to build the system. Design is an iterative process of either manually proposing or automatically synthesizing alternative solutions and then testing them with respect to the given constraints. The design is typically divided into many smaller subparts (following the well-known divide-and-conquer engineering approach) and each subpart is further divided, until the whole design is specified in terms of known primitive components.

The Verilog language provides the digital system designer with a means of describing a digital system at a wide range of levels of abstraction, and, at the same time, provides access to computer-aided design tools to aid in the design process at these levels. The language supports the early conceptual stages of design with its behavioral constructs, and the later implementation stages with its structural constructs. During the design process, behavioral and structural constructs may be mixed as the logical structure of portions of the design is designed. The description may be simulated to determine correctness, and some synthesis tools exist for automatic design. Indeed, the Verilog language provides the designer entry into the world of large, complex digital systems design.

The ability to describe a design is either in terms of the abstract behavior of the design or in terms of its actual logical structure. The behavioral description allows for early design activities to concentrate on functionality. When the behavior is agreed upon, then it becomes the specification for designing possibly several alternate structural implementations.

The Verilog language describes a digital system as a set of modules. Each of these modules has an interface to other modules as well as a description of its contents. A module represents a logical unit that can be described either by specifying its internal logical structure – for instance, describing the actual logic gates it is comprised of, or by describing its behavior in a program-like manner – in this case focusing on what the module does rather than on its logical implementation. These modules are then interconnected with nets, allowing them to communicate.

4.1.3. Design styles and abstraction levels in Verilog HDL

Verilog HDL, likes any other hardware description language, permits the designers to design a design in either bottom-up or top-down methodology.

A. Bottom-Up Design

The traditional method of electronic design is bottom-up. Each design is performed at the gate-level using the standard gates. With increasing complexity of new designs, this approach is nearly impossible to maintain. New systems consist of ASIC or microprocessors with a complexity of thousands of transistors. These traditional bottom-up designs have to give way to new structural, hierarchical design methods. Without these new design practices, it would be impossible to handle the new complexity.

B. Top-Down Design

The desired design-style of all designers is the top-down design. A real top-down design allows early testing, easy change of different technologies, structured system design, and offers many other advantages. But it is very difficult to follow a pure top-down design. Due to this fact, most designs are mix of both the methods, implementing some key elements of both design styles.

Verilog supports design at many different levels of abstraction. Three of them are very important:

a) Behavioral level:

This level describes a system by concurrent algorithms (behavioral). Each algorithm itself is sequential, which means it consists of a set of instructions that are executed one after the other. Functions, Tasks and Always blocks are the main elements. There is no regard to the structural realization of the design.

b) Register transfer level:

Designs using the register transfer level specify the characteristics of a circuit by operations and the transfer of data between the registers. An explicit clock is used. RTL design contains exact timing possibility; operations are scheduled to occur at certain times.

c) Gate level

Within the logic level, the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical

values (`0', `1', `X', `Z`). The usable operations are predefined logic primitives (AND, OR, NOT, etc. gates). If we require high speed and low power, then better to use gate level design to optimize the design. Some designs do full custom design, where logic is optimized at transistor level, like processors.

4.1.4. An example: s27 benchmark circuit described by Verilog HDL

There are two types of code (*Structural* and *Procedural*) in HDLs.

a) *Structural* is a verbal wiring diagram without storage, for example:

```
Assign a=b & c | d;
Assign d= e & (~c);
```

Here the order of the statements does not matter, changing e will change a.

b) *Procedural* that is used for circuits with storage, or containing conditional logic is as follows:

```
Always @ (posedge clk)    // Execute the next statement on every rising clock edge.
Count <=count+1;
```

Procedural code is written like c code and assumes every assignment is stored in memory. For synthesis, with flip-flop storage, this type of thinking generates too much storage. However, people prefer procedural code because it is usually much easier to write. For example, if and case statements are only allowed in procedural code. As a result, the synthesizers have been constructed which can recognize certain styles of procedural code as actually combinational.

s27 is a basic sequential circuit in ISCAS 89 benchmark circuits. There are different numbers of D flip-flops in every such sequential circuit as memory units. We first describe a D flip-flop (Figure 4.1) by Verilog HDL.

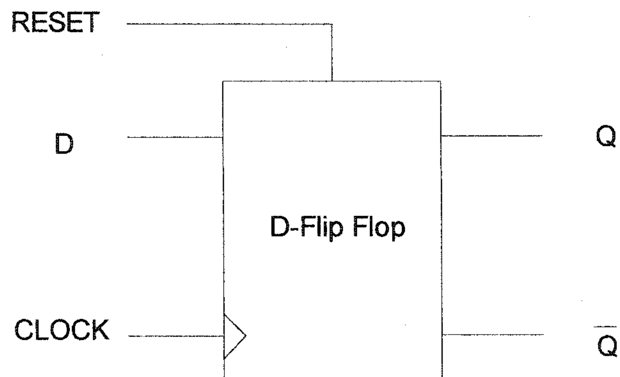


Figure 4.1. A D flip-flop.

The Verilog HDL description of the D flip-flop is given below:

```
module ndff(q,d,rst,clk);
  input clk,rst,d;
  output q;
  reg q;

  always @(posedge clk or posedge rst)
  if (rst==1) q<=0;
  else q<=d;
endmodule
```

We now present the schematic of s27 benchmark circuit in Figure 4.2.

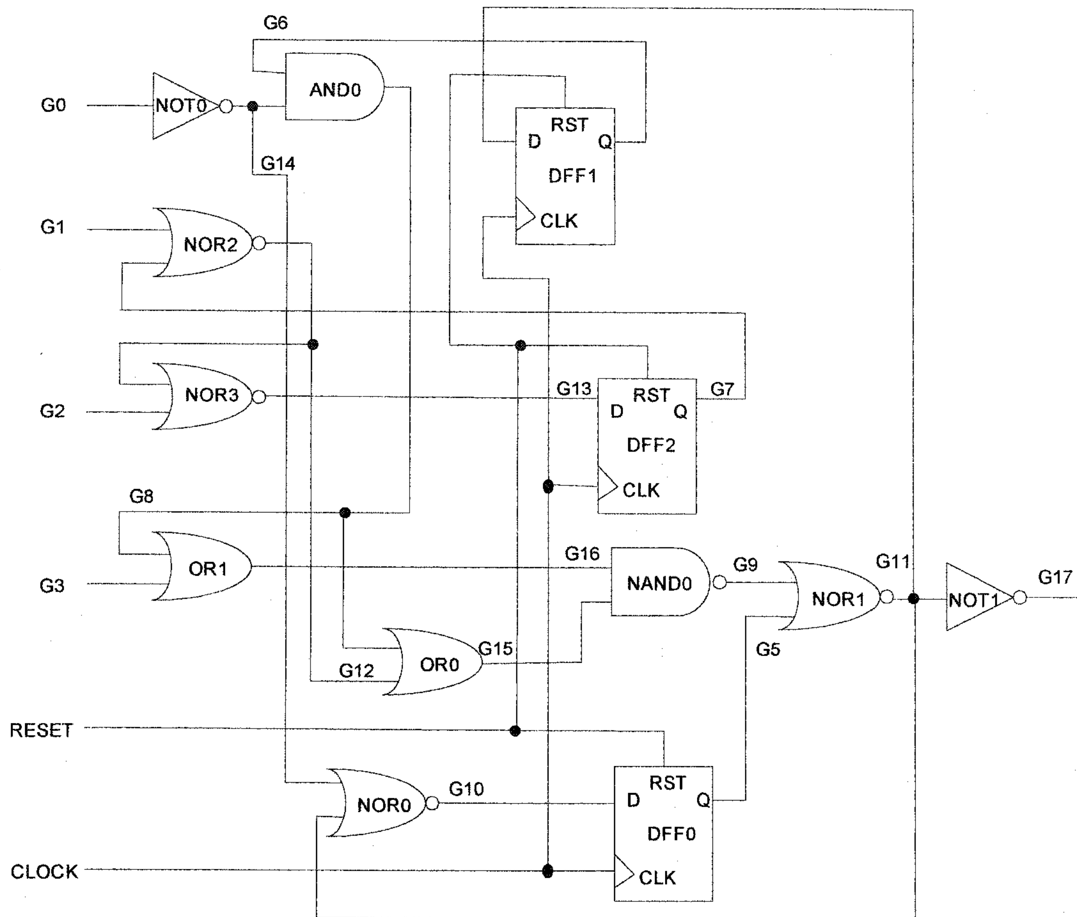


Figure 4.2. s27 gate-level schematic.

We use Verilog HDL to describe the s27 sequential circuit as given below.

```
# s27
# 6 inputs
# 1 outputs
# 3 D-type flipflops
# 2 inverters
# 8 gates (1 ANDs + 1 NANDs + 2 ORs + 4 NORs)

module ndff(rst,clk,q,d);
  input clk,rst,d;
  output q;
  reg q;

  always @(posedge clk or posedge rst)
  if (rst==1) q<=0;
```

```
  else q<=d;
endmodule

module s27(CK,rst,G0,G1,G17,G2,G3);
input CK,rst,G0,G1,G2,G3;
output G17;

  wire G5,G10,G6,G11,G7,G13,G14,G8,G15,G12,G16,G9;

  ndff DFF_0(rst,CK,G5,G10);
  ndff DFF_1(rst,CK,G6,G11);
  ndff DFF_2(rst,CK,G7,G13);
  not NOT_0(G14,G0);
  not NOT_1(G17,G11);
  and AND2_0(G8,G14,G6);
  or OR2_0(G15,G12,G8);
  or OR2_1(G16,G3,G8);
  nand NAND2_0(G9,G16,G15);
  nor NOR2_0(G10,G14,G11);
  nor NOR2_1(G11,G5,G9);
  nor NOR2_2(G12,G1,G7);
  nor NOR2_3(G13,G2,G12);

endmodule
```

After completing the description of s27 with Verilog HDL, we can start to compile the code and do simulation under Altera MAX PLUS II design environment.

4.2. Simulation Under MAX PLUS II Version 10.1

Figure 4.3 shows a simulating window for D flip-flop used in s27 sequential circuit.

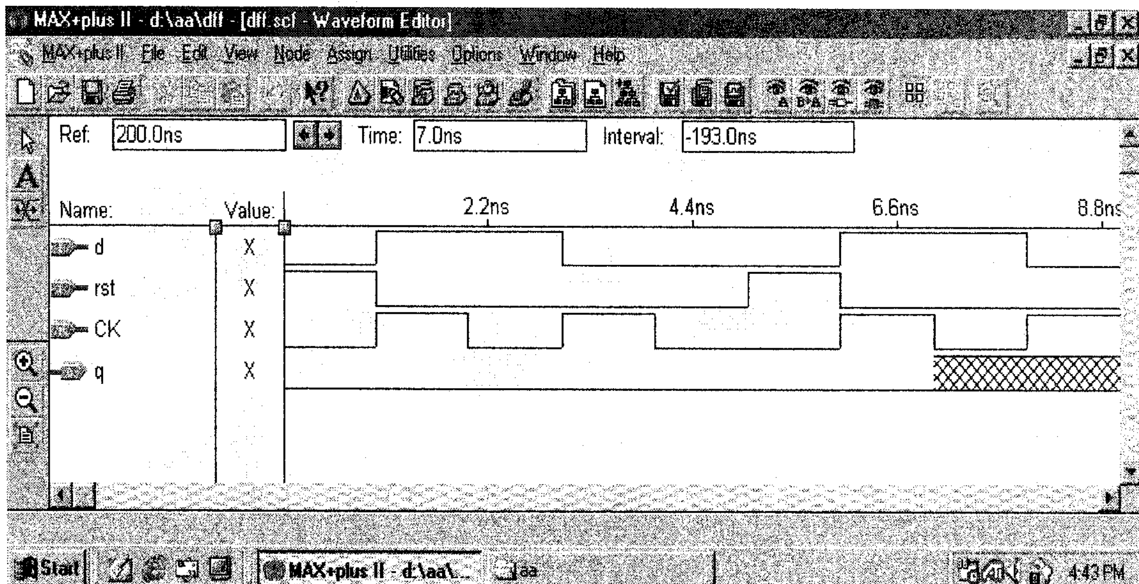


Figure 4.3. Simulation of a D flip-flop under Altera MAX PLUS II.

Altera MAX PLUS II development software is a fully integrated programmable logic design environment. This easy-to-use software supports the Altera ACEX, FLEX, MAX, and Classic programmable device families, and works in both PC and UNIX environments. Altera MAX PLUS II development software offers flexibility and performance, and allows seamless integration with industry-standard design entry, synthesis, and verification tools. Altera MAX PLUS II software is a comprehensive tool for the design, compilation, and simulation of digital circuit designs. Today, lots of designers like to develop, compile, verify, and program their digital circuit designs using Altera MAX PLUS II development software, which operates on personal computers (PCs) and engineering workstations.

Altera MAX PLUS II development software gives designers the flexibility to enter a design using all the major design entry methodologies, including:

- Verilog HDL
- VHDL
- Schematic capture
- Design entry utilizing megafunctions and the library of parameterized modules (LPM)
- Altera hardware description language (AHDL)

- Waveform

By giving designers control over the entry format and the ability to mix and match design entry methodologies, the Altera MAX PLUS II development software minimizes development time.

4.2.1. Altera MAX PLUS II design environment

The Altera multiple array matrix programmable logic user system (MAX PLUS II) provides a multi-platform, architecture-independent design environment that easily adapts to any specific design need. MAX PLUS II offers easy design entry, quick processing, and straightforward device programming.

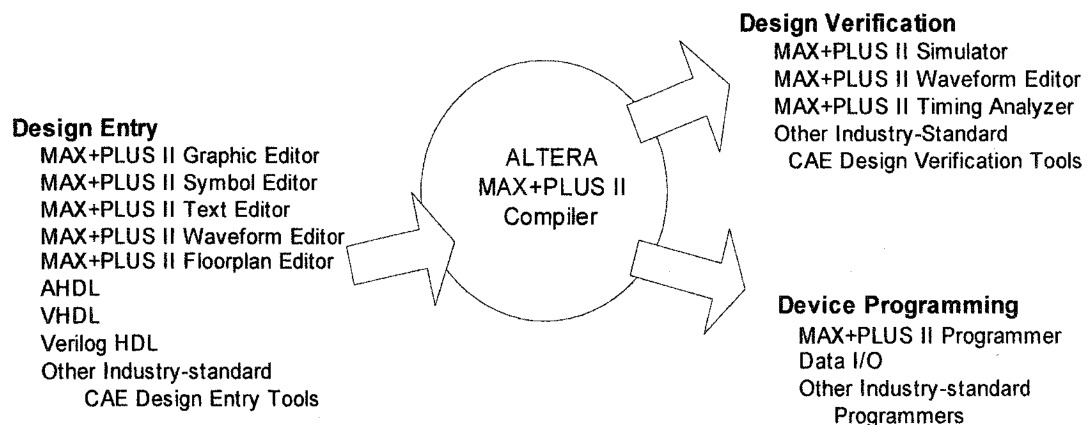


Figure 4.4. MAX PLUS II design environment.

MAX PLUS II development software, shown in Figure 4.4, is a fully integrated package for creating logic designs. MAX PLUS II offers a full spectrum of logic design capabilities: a variety of design entry methods for hierarchical designs, powerful logic synthesis, timing-driven compilation, partitioning, functional and timing simulation, linked multi-device simulation, timing analysis, automatic error location, and device programming and verification. MAX PLUS II both reads and writes Altera hardware description language (AHDL) files and standard EDIF netlist files, Verilog HDL files, VHDL files, and ORCAD

schematic files. In addition, MAX PLUS II reads Xilinx netlist files and writes standard delay format (SDF) files for a convenient interface to other industry-standard CAE software.

MAX PLUS II also offers a rich graphical user interface complemented with an illustrated, easy-to-use on-line help system. The complete MAX PLUS II system includes 11 fully integrated applications that take one through every step of creating a design (a logic design, including all subdesigns, is called a "project" in MAX PLUS II).

Many features and commands, such as opening files; entering device, pin, and logic cell assignments; and compiling the current project, are shared by many or all MAX PLUS II applications. The design editors (the graphic, text, and waveform editors) and auxiliary editors (the floorplan and symbol editors) also share numerous features. Each design editor allows one to perform similar tasks, such as finding a signal or symbol, in the same way. We can easily combine different types of design files in a hierarchical project, choosing the design entry format that works best for each functional block. A large library of Altera-supplied megafunctions and macrofunctions, including functions from the library of parameterized modules (LPM), provide a wide range of design entry options.

Designer can work with different MAX PLUS II applications simultaneously. For example, we can open multiple design files and transfer information among them while compiling or simulating another project; or, we can view an entire project hierarchy and move smoothly from one hierarchical level to another, while MAX PLUS II automatically starts the appropriate design editor for each file.

The MAX PLUS II compiler lies at the heart of the MAX PLUS II system, providing powerful project processing that one can customize to achieve the best possible silicon implementation of the project. Automatic error location and extensive documentation on error and warning messages make design modifications quick and easy. We can create output files in a variety of formats for functional, timing, and linked multi-device simulation; timing analysis; and device programming.

The superb integration of the MAX PLUS II software helps us maximize our efficiency and productivity, putting us in control of our logic design environment.

4.2.2. Simulation flow under Altera MAX PLUS II

When a circuit design is completed, we need to compile it for the chip on the Altera Board and simulate it through a sequence of inputs to verify this design. The process can be simplified as follows:

- Create a new design file or a hierarchy of multiple design files in any combination of the MAX PLUS II design editors (graphic, text, and waveform editors).
- Specify the top-level design file name as the project name.
- Assign a device family for the project. One can either allow the compiler to select a device for one or assign a specific device.
- Open the MAX PLUS II compiler window and choose the start button to compile the project. We also can turn on the timing SNF extractor module to create a netlist file for timing simulation and timing analysis.
- If the project compiles successfully, we can perform a simulation. To run a simulation, we must first create vector inputs in a simulator channel file (.scf) in the waveform editor or in a vector file (.vec) in the text editor. Then, we open the MAX PLUS II simulator window and choose the Start button to run the simulation. The simulator will calculate the circuit response (including circuit delays) for the output pins. These will automatically be shown on the Waveform displays. Also, we can get a Truth Table (.tbl) file from the simulation results.

4.2.3. MAX PLUS II applications

MAX PLUS II software consists of some application programs and the MAX PLUS II manager. Different design entry applications can be active simultaneously, allowing designers to switch between them with a click of the mouse or a menu command. At the same time, we can run one of the background applications - the compiler, simulator, timing analyzer, or programmer. Commands shared by the various applications function in the same way, making our logic design task easier.

We can describe the MAX PLUS II applications as follows (Figure 4.5):

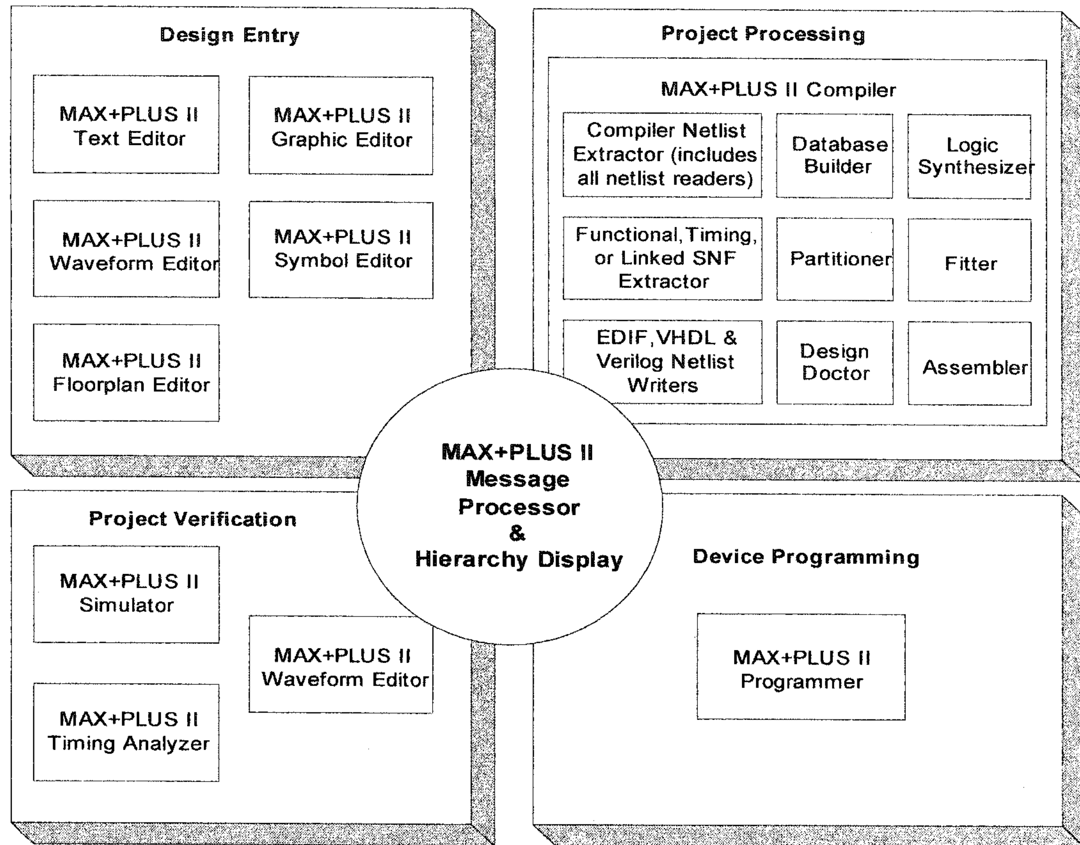


Figure 4.5. MAX PLUS II applications.

Hierarchy display: Displays the current hierarchy of files as a hierarchy tree with branches that represent subdesigns: whether a design file is a schematic, text, or waveform design; which files are currently open; and which user-editable ancillary files are available for the project. We can also directly open or close one or more files in a hierarchy tree and enter resource assignments for them.

Graphic editor: Lets designer enter a schematic logic design. While the Altera-provided primitives, megafunctions, and macrofunctions serve as designer's basic building blocks, designer can also use custom symbols.

Symbol editor: Allows designer to edit existing symbols and create new ones.

Text editor: The text editor lets designer create and edit text-based logic design files written in AHDL, VHDL, and Verilog HDL. With the text editor, designer can also create, view, and edit other ASCII files used with MAX PLUS II applications. Although designer can create HDL files with other text editors, the MAX PLUS II text editor allows designer to

take advantage of context-sensitive help, syntax coloring, and AHDL, VHDL, and Verilog HDL templates.

Waveform editor: Serves a dual role, as a design entry tool and as a tool for entering test vectors and viewing simulation results.

Floorplan editor: Lets designer assign logic to physical device pin and logic cell resources in a graphical environment. Designer can edit pin placements in a device package view and assign signals to individual logic cells in a more detailed logic array block (LAB) view. Designer can also view the results of the last compilation.

Compiler: Processes logic projects targeted for Altera device families. It performs most tasks automatically. However, designer can customize all or part of the compilation process.

Simulator: Enables designer to test the logical operation and internal timing of designer's logic circuit. Functional simulation, timing simulation, and linked multi-device simulation are available.

Timing analyzer: Analyzes the performance of designer's logic circuit after it has been synthesized and optimized by the compiler.

Programmer: Lets designer program, configure, verify, examine, and test Altera devices.

Message processor: Displays error, warning, and information messages on the status of designer's project and allows user to locate the source of a message automatically in the original design file(s), ancillary file(s), and assignments floorplan.

Chapter 5

DESIGN AND IMPLEMENTATION

5.1. General Structure in Sequential Circuit Testing

In this section, we will introduce the general structure and methodology of testing synchronous cores-based sequential circuits.

Figure 5.1 shows the system architecture that can be used to realize the testing scheme in hardware.

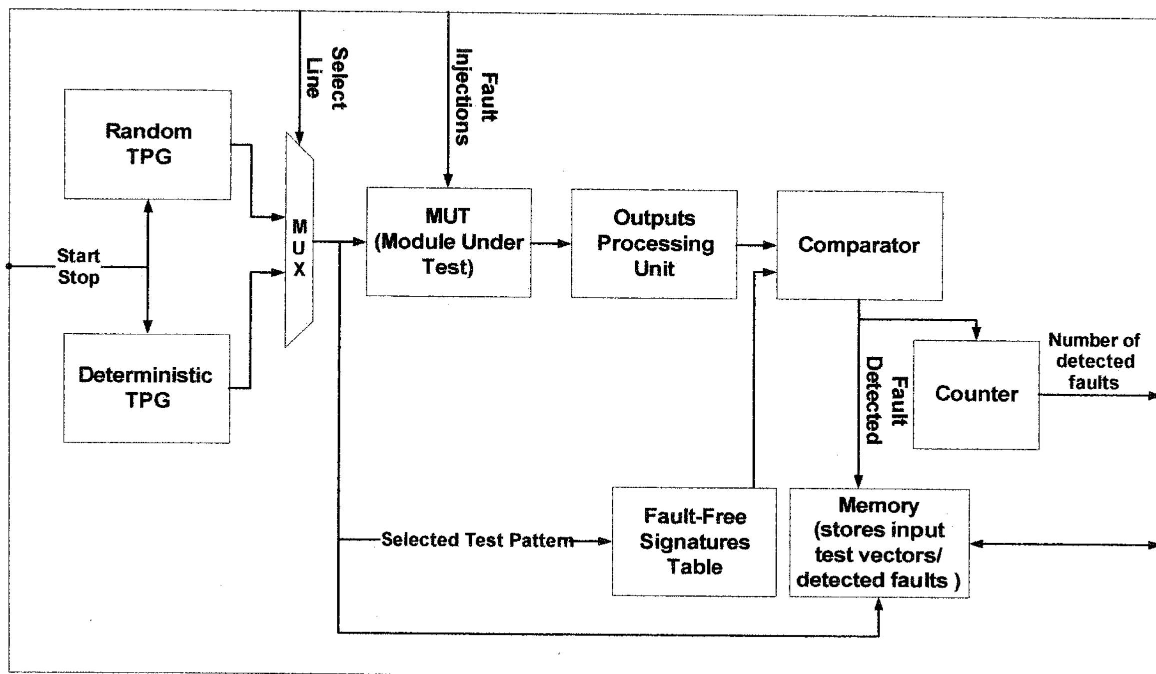


Figure 5.1. The testing scheme in hardware at system level.

In the thesis, we will focus on developing a testing system for synchronous cores-based sequential circuits by full software simulation rather than by using hardware.

Figure 5.2 shows a general design flow of implementation of the test methodology.

We implemented the fault simulation approach using C programming language on an IBM-compliant personal computer (with Pentium III, 1GHz, 256 MB of RAM). In our implementation, we will use system function of C to call MAX PLUS II for compilation and simulation of an MUT (module under test) on the background. An MUT will be represented by Verilog HDL gate-level description. The executable file and the circuits in ISCAS 89 full-scan sequential benchmark circuits will be stored in the same file fold.

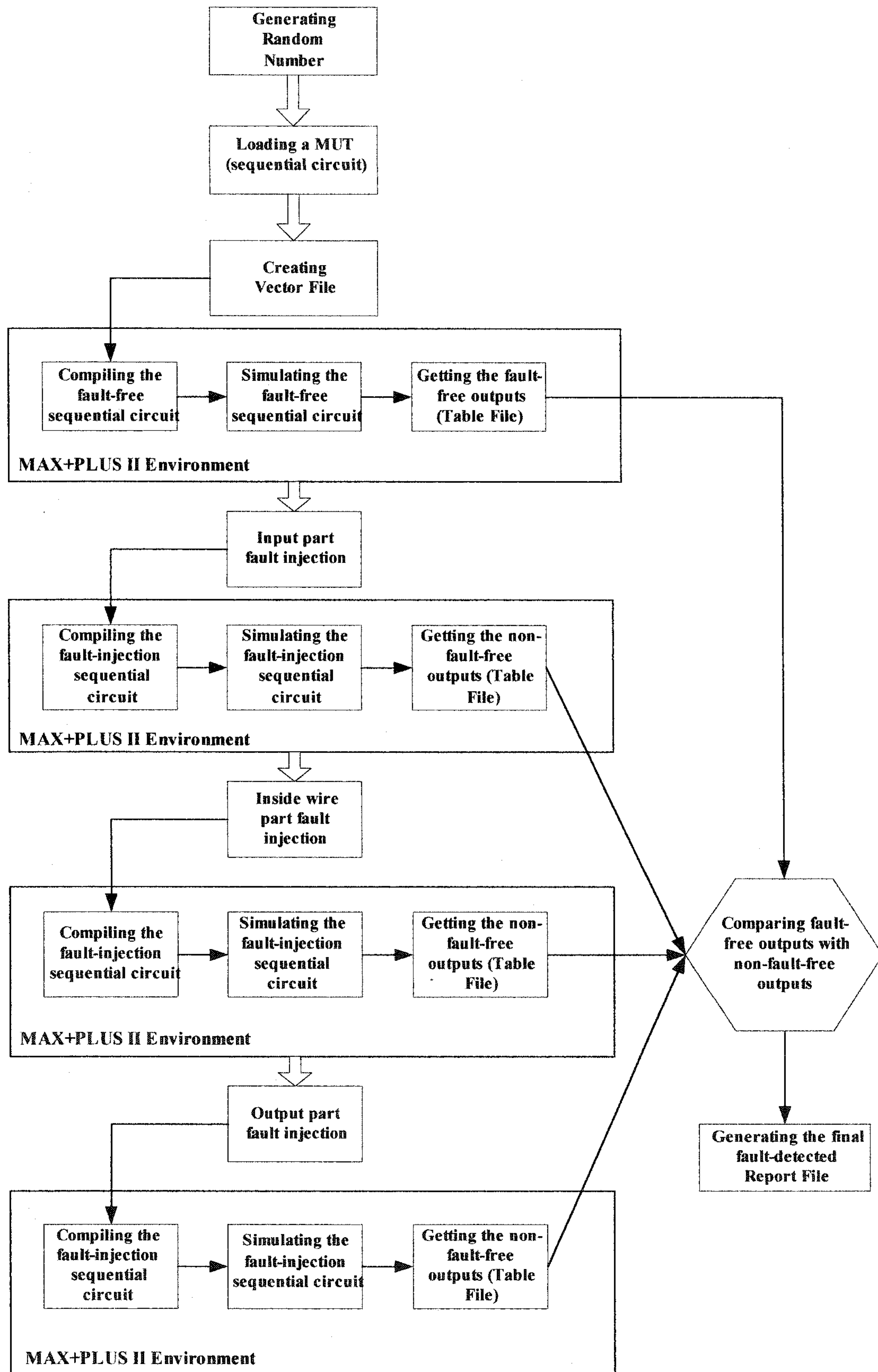


Figure 5.2. Design flow of the sequential circuit testing method.

5.2. Initialization

The goal in the initialization phase is to generate random signal, create input vector file for MUT and reset all flip-flops that are in the unknown states.

5.2.1. Random Signal Generator

In BIST techniques, exhaustive, pseudoexhaustive, pseudorandom, or reduced test patterns are used because of the ease of their generation and storage on-chip. Many practical circuits need few test patterns for full coverage of single stuck-faults. Using practical algorithms (such as PODEM, FAN), these reduced test sets can be generated on-chip at a low hardware cost with high fault coverage.

5.2.1.1. Hardware implementation of random signal generator

The autonomous linear feedback shift register (ALFSR) can be used to generate pseudorandom test vectors. An ALFSR is a serial connection of D flip-flops with no external inputs and Exclusive-OR (XOR) gates providing the feedback. A four-stage ALFSR is shown in Figure 5.3.

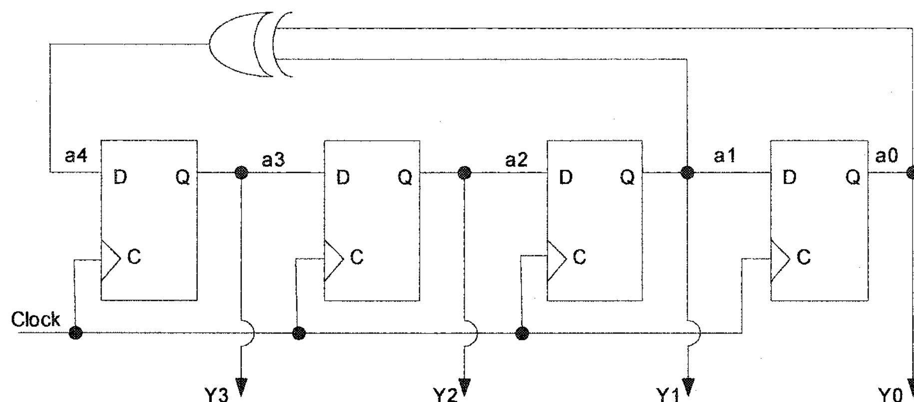


Figure 5.3. An ALFSR structure.

Autonomous linear feedback shift registers make good pseudorandom pattern generators. When the outputs of the flip-flops are loaded with a seed value and when the ALFSR is clocked, it will generate a pseudorandom pattern of 1s and 0s. The only signal necessary to generate the test patterns is the clock.

5.2.1.2. Software simulation of random signal generator

In the thesis, we will implement random signal generator (RNG) by software simulation. Then, according to a specified sequential circuit, we will put the random number into a .vec file as the format of Altera MAX PLUS II development software. Finally, we get the vector file and use the vector as inputs for the sequential circuit. We implement the random signal generator using C programming language.

The random signal generator is called as random_number() function. It is a portable, fast and good pseudorandom number generator. When we need some random number, we simply call the random_ number() functions. Initializing the function with a certain seed will produce exactly the same series of random numbers on all platforms. The random_number() functions will return high quality equally distributed random numbers.

In the following we give the pseudocode description of the algorithm of the random signal generator.

// The pseudocode description of the modified RNG algorithm that generates the sets of random numbers

// This program seeds the random number generator with the time, then displays the random integers as needed.

// Define a function random_number () for random number generator

unsigned long int random_number(unsigned long int seed)

{

// Determing the dimensions (size) of the int parameter module

const unsigned long int module ← int1;

// Determing the dimensions (size) of the int parameter multiplier

```
  const unsigned long int multiplier ←int2;
// The value of seed can be initialized to any value if preferred
  static unsigned long int ran ← seed;
// Initial value is used only at the first time
// Generating new, modified ran number using the algorithm
  ran←(ran * multiplier) % module;
  return ran;
}
// In main function, call function random_number ()
main
{
// Call srand() function,set a random starting point, initialize random generator
// Seed the random number generator with current time.
srand (time(NULL));
// Call rand() function to generate the seed for random_number () function.
// Seed for random number generation
seed ← rand();
// Generating random numbers between 0 to 1 and store into gen
gen ← random_number(seed) % 2;
}
```

5.2.2. Input vector file

In Altera MAX PLUS II, vector file is used as the source of input vectors for simulation. A vector specifies the logic levels for an individual node within a project. The simulator uses vectors to simulate the behavior of the current MUT.

Vectors for simulation and functional testing can be defined in vector files (.vec). Functional testing vectors can also be stored in programming files.

The vector file provides an ASCII text format for specifying simulation input conditions and the nodes to be simulated. A vector file can also be used to create a waveform design file

for waveform design entry. Vectors that have been saved in the programming file can also be used for functional testing.

A vector file that is used for functional testing must contain all input vector logic levels, and expected output logic levels that are typically derived from a simulation or from a previously tested device.

The simulator looks for a vector file or a simulator channel file, when we execute the SIMULATE command. It automatically loads the newest vector file or SCF in the project directory with the same filename as the project. When we enter VECTOR, the simulator automatically generates an SCF for the file. Expected output logic levels are optional. They may be of any type: user-defined expected outputs, outputs from other simulations, or actual functional device outputs (e.g., in a table file created during a previous simulation).

The following example shows a sample vector file, including a separate pattern section for optional expected output values.

```
% units default to ns. %
% Specify start and stop time of simulation. %

START 0 ;
STOP 1000 ;

% Time between steps %
INTERVAL 100 ;
INPUTS CLOCK ;

PATTERN
0 1 ;               % relative vector values %

INPUTS DATAINX DATAINY ;
PATTERN             % test every combination of %
                    % DATAINX and DATAINY %
0>     0 0
220>   1 0
320>   1 1          % absolute time vector values %
570>   0 1
720>   1 1
;

INPUTS CLEAR ;
PATTERN
```

```
0>     1
100>   0
;

OUTPUTS SERSUM CIN COUT  ;
PATTERN          % check output at every Clock pulse %
= X X X
= 0 0 0          % relative time vector values %
= 0 0 0
= 1 0 0
= 0 0 1
= 0 1 1
= 1 1 1 ;
```

In the following we give the pseudocode description of the algorithm of forming a vector file for the MUT.

// The pseudocode description of the algorithm that makes a vector file for a sequential circuit according to Altera MAX PLUS II format.

```
// Create a new text file for writing by using fopen() function.
// Get the number of inputs from MUT
// Get all input names of MUT
// Construct the head part of vector file by fprintf() function.
// Generating the test pattern for inputs
  for (int i = 0; i < number of test patterns ; ++i)
   {
     for (int count = 0; count < number of inputs; ++count)
      {
        // Call random number generator to generate the random number
        // Add the reset signal for every test pattern
        // Add the clock signal for every test pattern
        }
     }
 // Get the number of outputs from MUT
 // Get all output names of MUT
```

// Construct the end part of vector file by fprintf() function

// Close and save the vector file.

5.3. Fault Injection and Fault Simulation

Now we will implement the most important and complex section in our implementation – fault injection and simulation.

5.3.1. Hardware fault injection

The hardware fault injection technique was devised in order to iteratively inject faults to every mutually exclusive wire, and to test for both the stuck-at-0 and stuck-at-1 faults. Figure 5.4 shows the fault injection scheme in hardware.

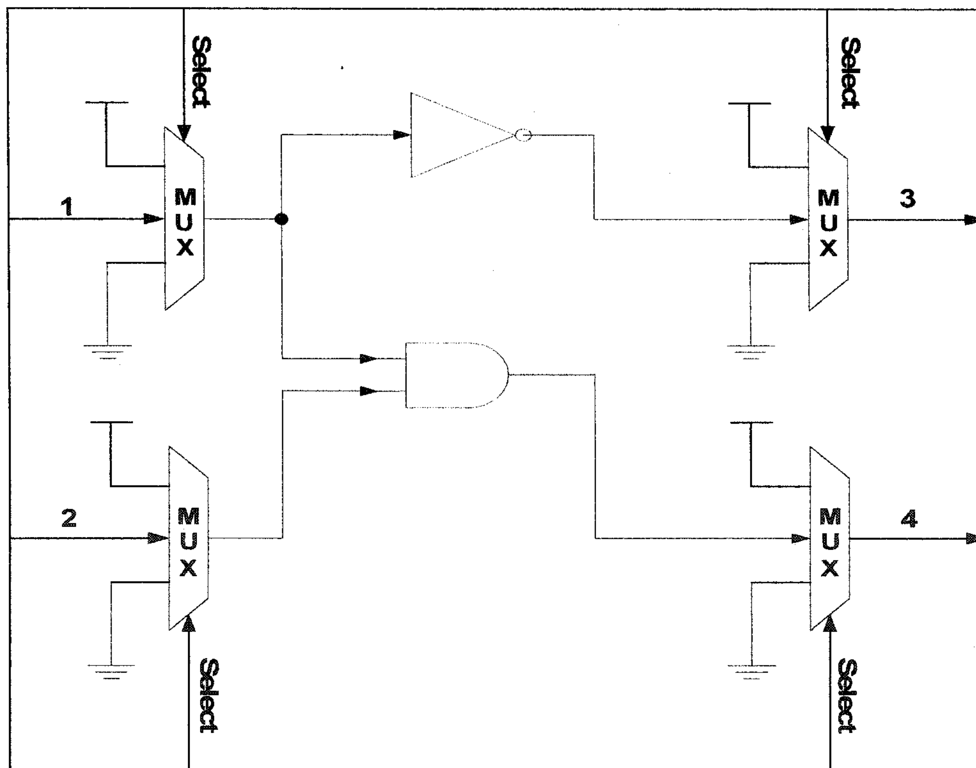


Figure 5.4. The fault injection scheme in hardware.

In fault injection scheme, every mutually exclusive wire now has a multiplexer (MUX) introduced within it, which allows us to either run the wire as is, or inject stuck-at-0 or stuck-

at-1 faults. If the value of the select signals of any multiplexer are at "00", then we will run the wire as is, while if the values are at "01", then we will inject a stuck-at-1 fault indicated by the logical 1 value coming into the multiplexer, and if the values are at "10", then we will inject a stuck-at-o fault indicated by the logical 0 value coming into the multiplexer. Finally, if the values are at "11", then we will again assume normal operation of the wire.

5.3.2. Software fault injection

In the implementation, we will focus on software fault injection, that is, we will simulate the above part – hardware fault injection scheme with entire software method. The software fault injection will include three parts: input part fault simulation, inside wire fault simulation and output part fault simulation.

5.3.2.1. Input part fault simulation

In the following we give the pseudocode description of the algorithm of input part fault simulation.

// The pseudocode description of the algorithm that perform the input part fault simulation for a sequential circuit by compilation and simulation under Altera MAX PLUS II development environment.

```
// Set a flag to indicate the injecting fault to be "1" or "0".
  bz=-1;
// Perfoming fault injection one by one from the fist input line to the last input line
 for (pin=1;pin< number of inputs-1;pin++)
 {
   if (the flag for injecting fault<0)
   set the injecting fault as 0
   if (the flag for injecting fault>0)
   set the injecting fault as 1
```

```
//Open the .vecbak (backup file of vector file) text file for reading
// Generating new, modified .vector file and processing them:
        while ( .vecbak text file is not empty )
        {
          // Read a line from .vecbak text file and find the key character ">";
          if ( the key character be found )
          // Modify the line and do the fault injection for the current input pin
          // Store this modified line to new modified vector text file;
          if ( the key character not be found )
          // Copy this text line to new modified vector file;
        }
        // Close the .vecbak text file;
       // Close and save the new vector file for future using.

// Invoking command interpreter to execute a Compile command of ALTER MAXPLUS II to
compile the MUT and vector file on the background

//  Invoking command interpreter to execute a Simulate command of ALTER MAXPLUS II to
simulate the MUT using vector file as input file on the background

// Call outputs comparing and recording function
// Modifying the flag of injecting fault then performing a new fault injection for the current
input.
}
// Recover the original vector file and end input fault injection and testing.
```

5.3.2.2. Inside wire fault simulation

In the following we give the pseudocode description of the algorithm of inside wire fault simulation.

```
// The pseudocode description of the algorithm that perform the inside wire fault simulation
for a sequential circuit by compilation and simulation under Altera MAX PLUS II
development environment.

// Set a flag to indicate the injecting fault to be "1" or "0".
  bz=-1;
// Set a counter for recording the current wire

// Perfoming fault injection one by one from the fist inside wire to the last inside wire
 for (wir=0;wir<number of inside wires;wir++)
 {
   if (the flag for injecting fault<0)
   set the injecting fault as 0
   if (the flag for injecting fault>0)
   set the injecting fault as 1

 // Getting the name of current inside wire line from MUT verilog file (.v text)
 // Open the .vlgbak (backup file of MUT verilog file) text file for reading
 // Generating new, modified MUT verilog file and processing them:

        while ( .vlgbak text file is not empty )
        {
          // Read a line from vecbak text file
          // Find the inside wire that should be injected a fault
           if ( the inside wire be found )
           // Modify the inside wire and do the fault injection for the current inside wire
           // Store this modified line to new modified MUT verilog text file;
           if ( the inside wire not be found )
```

// Copy this text line to new modified vector file;

}

// Close the .vlgbak text file;

// Close and save the new MUT verilog text file for future using.

// Invoking command interpreter to execute a Compile command of ALTER MAXPLUS II to compile the MUT and vector file on the background

// Invoking command interpreter to execute a Simulate command of ALTER MAXPLUS II to simulate the MUT using vector file as input file on the background

// Call outputs comparing and recording function
// Modifying the flag of injecting fault then performing a new fault injection for the current inside wire.

}

// Recover the original MUT verilog file and end inside wire fault injection and testing.

5.3.2.3. Output part fault simulation

In the following we give the pseudocode description of the algorithm of output part fault simulation.

// The pseudocode description of the algorithm that perform the output part fault simulation for a sequential circuit by compilation and simulation under Altera MAX PLUS II development environment.

// Set a flag to indicate the injecting fault to be "1" or "0".

bz=-1;

// Set a counter for recording the current output pin

```
// Perfoming fault injecting and testing by one from the fist output pin to the last output pin
 for (pin=0;pin<number of output pins;pin++)
 {
   if (the flag for injecting fault<0)
   set the injecting fault as 0
   if (the flag for injecting fault>0)
   set the injecting fault as 1

//Open the .tbak (backup file of table file) text file for reading

        while ( .tbak text file is not empty )
        {
          // Read a line from .vecbak text file
          // Get the value of the current output pin;
           if ( the value does not equal  the injecting fault )
          // One fault is found and record the current test pattern
         // Terminate the execution of the nearest while loop
         }

 // Modifying the flag of injecting fault then performing a new fault injection and testing for
 the current output pin.
 }

 // Close the .tlgbak text file and end output pin fault injection and testing.
```

5.4. Comparison of Results and Reports

An ASCII-format Table File (with the extension .tbl) can be used to record the results of current simulation under Altera MAX PLUS II development environment.

The Table File reports logic levels for the nodes and groups in the Vector File or Simulator Channel File (SCF) specified with Inputs/Outputs as the source of input vectors for simulation or functional testing. The reported nodes consist of all nodes and groups in the SCF, or the nodes and groups listed in the Vector File's Inputs and Outputs.

The following example shows a sample Table File.

```
%
MAX+plus II 10.1    Date: 08/25/2003 20:02:11
File Generated From: s344.scf

Copyright (C) 1988-2001 Altera Corporation.
%

INPUTS START B0 B1 B2 B3 A0 A1 A2 A3 rst CK ;
OUTPUTS P4 P5 P6 P7 P0 P1 P2 P3 CNTVCON2 CNTVCO2 READY ;
UNIT ns ;
RADIX HEX ;
PATTERN
%                                                         C     %
%                                                         N C   %
%                                                         T N   %
%      S                                                  V T R %
%      T                                                  C V E %
%      A                       r                          O C A %
%      R B B B B A A A A s C   P P P P P P P P N O D %
%      T 0 1 2 3 0 1 2 3 t K   4 5 6 7 0 1 2 3 2 2 Y %

  0.0> 0 0 1 1 0 1 1 1 1 1 1 = 1 1 1 1 1 1 1 1 1 0 0
  1.0> 0 0 1 1 0 1 0 0 1 0 0 = 1 1 1 1 1 1 1 1 1 0 0
  2.0> 0 0 1 1 0 1 0 0 1 0 1 = 1 1 1 1 1 1 1 1 1 0 0
  2.1> 0 0 1 1 0 1 0 0 1 0 1 = 1 1 1 1 0 1 1 0 1 0 0
  3.0> 1 1 0 1 0 0 0 0 0 0 0 = 1 1 1 1 0 1 1 0 1 0 0
  4.0> 1 1 0 1 0 0 0 0 0 0 1 = 1 1 1 1 0 1 1 0 1 0 0
  4.1> 1 1 0 1 0 0 0 0 0 0 1 = 0 0 0 0 1 1 0 1 1 0 0
  5.0> 0 1 0 1 0 0 1 0 0 0 0 = 0 0 0 0 1 1 0 1 1 0 0
  6.0> 0 1 0 1 0 0 1 0 0 0 1 = 0 0 0 0 1 1 0 1 1 0 0
  6.1> 0 1 0 1 0 0 1 0 0 0 1 = 0 0 0 0 1 0 1 0 1 0 0
  7.0> 1 0 0 1 0 0 0 1 0 0 0 = 0 0 0 0 1 0 1 0 1 0 0
  8.0> 1 0 0 1 0 0 0 1 0 0 1 = 0 0 0 0 1 0 1 0 1 0 0
  8.1> 1 0 0 1 0 0 0 1 0 0 1 = 0 0 0 0 0 1 0 0 1 0 0
  9.0> 0 1 1 0 0 1 1 0 0 0 0 = 0 0 0 0 0 1 0 0 1 0 0
 10.0> 0 1 1 0 0 1 1 0 0 0 1 = 0 0 0 0 0 1 0 0 1 0 0
 10.1> 0 1 1 0 0 1 1 0 0 0 1 = 0 0 0 0 1 1 0 0 1 0 0
 11.0> 1 1 1 1 0 0 0 1 0 0 0 = 0 0 0 0 1 1 0 0 1 0 0
 12.0> 1 1 1 1 0 0 0 1 0 0 1 = 0 0 0 0 1 1 0 0 1 0 0
 12.1> 1 1 1 1 0 0 0 1 0 0 1 = 0 0 0 0 1 0 0 1 1 0 0
 13.0> 0 0 0 0 0 1 0 1 1 0 0 = 0 0 0 0 1 0 0 1 1 0 0
 14.0> 0 0 0 0 0 1 0 1 1 0 1 = 0 0 0 0 1 0 0 1 1 0 0
```

```
14.1> 0 0 0 0 0 1 0 1 1 0 1 = 0 0 0 0 0 0 0 0 1 0 0
15.0> 0 0 1 1 0 0 0 1 0 0 0 = 0 0 0 0 0 0 0 0 1 0 0
16.0> 0 0 1 1 0 0 0 1 0 0 1 = 0 0 0 0 0 0 0 0 1 0 0
17.0> 0 0 0 0 1 0 1 1 0 0 0 = 0 0 0 0 0 0 0 0 1 0 0
18.0> 0 0 0 0 1 0 1 1 0 0 1 = 0 0 0 0 0 0 0 0 1 0 0
19.0> 0 0 1 0 1 0 0 1 1 0 0 = 0 0 0 0 0 0 0 0 1 0 0
20.0> 0 0 1 0 1 0 0 1 1 0 1 = 0 0 0 0 0 0 0 0 1 0 0
21.0> 0 0 0 0 0 0 1 1 0 0 0 = 0 0 0 0 0 0 0 0 1 0 0
22.0> 0 0 0 0 0 0 1 1 0 0 1 = 0 0 0 0 0 0 0 0 1 0 0
22.1> 0 0 0 0 0 0 1 1 0 0 1 = 0 0 0 0 0 0 0 0 1 0 1
23.0> 1 1 0 1 0 1 0 0 0 0 0 = 0 0 0 0 0 0 0 0 1 0 1
24.0> 1 1 0 1 0 1 0 0 0 0 1 = 0 0 0 0 0 0 0 0 1 0 1
24.1> 1 1 0 1 0 1 0 0 0 0 1 = 0 0 0 0 0 0 0 0 1 0 0
25.0> 1 1 1 1 1 1 0 0 1 0 0 = 0 0 0 0 0 0 0 0 1 0 0
26.0> 1 1 1 1 1 1 0 0 1 0 1 = 0 0 0 0 0 0 0 0 1 0 0
;
```

By comparing two simulation results of MUTs (fault-injected and fault-free), we can detect those faults in the MUT.

In the following we give the pseudocode description of the algorithm of comparison of results and generate a fault file for MUT.

// The pseudocode description of the algorithm that perform comparison of results and generate a fault file for a sequential circuit by compilation and simulation under Altera MAX PLUS II development environment.

// Open the .tbl text file for reading
// Open the .tbak (backup file of a fault-free table file) text file for reading
// Set counter r to indicate the number of current line.

```
while ( .tbl text file is not empty )
{
  // Read a line from .tbl text file
  // Get the r line of test pattern and output   in .tbl text file
     // Store the current line to string1
  while ( .tbak text file is not empty )
  {
```

```
// Read a line from the .tbak text file
// Get the r line of test pattern and output  in the .tbak text file
  // Store the current line to string2
  // Compare the string1 and string2
  if ( different  between string1 and string2)
  // One fault is found and record the current test pattern and output
  // Terminate the execution of the nearest while loop
   break;
  }
  // Set counter r to indicate the next line
}
  // Create a fault report file.
        report = fopen(rpt, "a");
  // Construct the head part of fault report file
// Form main fault report part according the records from results comparison.
// Close and save the fault report and end Comparison of Results and Reports.
```

Chapter 6

TEST CASES AND EXPERIMENTAL RESULTS

This chapter of the thesis discusses simulation experiments on ISCAS 89 full-scan sequential benchmark circuits based on our use of Verilog HDL under MAX PLUS II development environment. Many of the details of implementation will be provided below considering s298 sequential benchmark circuit. We will use deterministic compacted test inputs as well as pseudorandom test vectors in the implementation in order to compare and discuss our simulation results.

6.1. Generating Random Signal Input Vector File of s298 Circuit

Initially, we will create a new file fold in our PC and put the main executable file SqTest_1.exe and s298.v (Verilog Design File) into this file fold. Then we will start to run the executable file SqTest_1.exe and input s298 as a name of MUT, which will create a random signal input vector as follows:

```
START 0 ;
INTERVAL 1 ;
INPUTS G0 G1 G2 rst CK;
PATTERN

0> 1 1 0 0 0
1> 1 1 0 0 1
2> 1 1 0 0 0
3> 1 1 0 0 1
4> 0 0 1 0 0
5> 0 0 1 0 1
6> 1 0 1 0 0
7> 1 0 1 0 1
8> 1 0 1 0 0
9> 1 0 1 0 1
10> 1 1 1 0 0
11> 1 1 1 0 1
12> 1 0 1 0 0
13> 1 0 1 0 1
14> 1 1 0 0 0
15> 1 1 0 0 1
16> 0 1 1 0 0
17> 0 1 1 0 1
18> 0 1 0 0 0
19> 0 1 0 0 1
20> 0 0 0 0 0
21> 0 0 0 0 1
22> 0 1 0 0 0
23> 0 1 0 0 1
24> 1 1 1 0 0
25> 1 1 1 0 1
26> 0 1 1 0 0
27> 0 1 1 0 1
28> 0 1 0 0 0
29> 0 1 0 0 1
30> 1 0 1 0 0
31> 1 0 1 0 1
32> 0 1 0 0 0
33> 0 1 0 0 1
34> 0 1 1 0 0
35> 0 1 1 0 1
36> 1 1 1 0 0
37> 1 1 1 0 1
38> 0 0 1 0 0
39> 0 0 1 0 1
40> 1 0 1 0 0
41> 1 0 1 0 1
42> 0 0 0 0 0
43> 0 0 0 0 1
44> 1 1 1 0 0
45> 1 1 1 0 1
46> 0 1 0 0 0
47> 0 1 0 0 1
48> 1 0 1 0 0
49> 1 0 1 0 1
50> 0 0 0 0 0
51> 0 0 0 0 1
52> 1 0 0 0 0
53> 1 0 0 0 1
```

```
54> 0 0 0 0 0
55> 0 0 0 0 1
56> 1 0 1 0 0
57> 1 0 1 0 1
58> 1 1 1 0 0
59> 1 1 1 0 1
60> 1 1 1 0 0
61> 1 1 1 0 1
62> 0 0 0 0 0
63> 0 0 0 0 1
64> 0 1 0 0 0
65> 0 1 0 0 1
66> 0 0 1 0 0
67> 0 0 1 0 1
68> 0 1 1 0 0
69> 0 1 1 0 1
70> 0 0 0 0 0
71> 0 0 0 0 1
72> 1 1 0 0 0
73> 1 1 0 0 1
74> 1 1 0 0 0
75> 1 1 0 0 1
76> 0 0 1 0 0
77> 0 0 1 0 1
78> 0 1 0 0 0
79> 0 1 0 0 1
80> 1 0 1 0 0
81> 1 0 1 0 1
82> 0 0 1 0 0
83> 0 0 1 0 1
84> 0 0 1 0 0
85> 0 0 1 0 1
86> 0 0 1 0 0
87> 0 0 1 0 1
88> 0 1 0 0 0
89> 0 1 0 0 1
90> 1 1 1 0 0
91> 1 1 1 0 1
92> 0 1 0 0 0
93> 0 1 0 0 1
94> 1 0 1 0 0
95> 1 0 1 0 1
96> 1 0 1 0 0
97> 1 0 1 0 1
98> 1 0 0 0 0
99> 1 0 0 0 1
100> 1 0 0 0 0
101> 1 0 0 0 1
102> 0 1 0 0 0
103> 0 1 0 0 1
104> 0 1 0 0 0
105> 0 1 0 0 1
106> 1 0 0 0 0
107> 1 0 0 0 1
108> 0 0 1 0 0
109> 0 0 1 0 1
110> 1 1 1 0 0
111> 1 1 1 0 1
112> 1 0 0 0 0
113> 1 0 0 0 1
114> 0 1 1 0 0
115> 0 1 1 0 1
116> 0 0 0 0 0
117> 0 0 0 0 1
118> 0 1 1 0 0
119> 0 1 1 0 1
120> 1 0 1 0 0
121> 1 0 1 0 1
122> 1 1 1 0 0
123> 1 1 1 0 1
124> 1 1 1 0 0
125> 1 1 1 0 1
126> 0 0 0 0 0
127> 0 0 0 0 1
128> 1 1 0 0 0
129> 1 1 0 0 1
130> 1 0 1 0 0
131> 1 0 1 0 1
132> 1 0 0 0 0
133> 1 0 0 0 1
134> 1 1 1 0 0
135> 1 1 1 0 1
136> 0 0 0 0 0
137> 0 0 0 0 1
138> 1 0 1 0 0
139> 1 0 1 0 1
140> 1 0 0 0 0
141> 1 0 0 0 1
142> 0 1 1 0 0
143> 0 1 1 0 1;

OUTPUTS G117 G132 G66 G118 G133 G67;
```

6.2. Generating Fault-Free Output File of s298 Circuit

Since we are first simulating a fault-free sequential circuit, we generate a fault-free output file – s298.tbl, as given below.

```
% MAX PLUS II 10.1   Date: 11/23/2003 21:35:18
File Generated From: s298.scf
%
INPUTS G0 G1 G2 rst CK;
OUTPUTS G117 G132 G66 G118 G133 G67;
UNIT ns;
RADIX HEX;
PATTERN
```

```
%                 G G   G G   %
%           r     1 1 G 1 1 G %
%     G G G s C   1 3 6 1 3 6 %
%     0 1 2 t K   7 2 6 8 3 7 %

 0.0> 1 1 0 0 0 = 0 0 0 0 0 0
 1.0> 1 1 0 0 1 = 0 0 0 0 0 0
 1.1> 1 1 0 0 1 = 1 0 0 0 0 1
 2.0> 1 1 0 0 0 = 1 0 0 0 0 1
 3.0> 1 1 0 0 1 = 1 0 0 0 0 1
 4.0> 0 0 1 0 0 = 1 0 0 0 0 1
 5.0> 0 0 1 0 1 = 1 0 0 0 0 1
 6.0> 1 0 1 0 0 = 1 0 0 0 0 1
 7.0> 1 0 1 0 1 = 1 0 0 0 0 1
 8.0> 1 0 1 0 0 = 1 0 0 0 0 1
 9.0> 1 0 1 0 1 = 1 0 0 0 0 1
10.0> 1 1 1 0 0 = 1 0 0 0 0 1
11.0> 1 1 1 0 1 = 1 0 0 0 0 1
12.0> 1 0 1 0 0 = 1 0 0 0 0 1
13.0> 1 0 1 0 1 = 1 0 0 0 0 1
14.0> 1 1 0 0 0 = 1 0 0 0 0 1
15.0> 1 1 0 0 1 = 1 0 0 0 0 1
16.0> 0 1 1 0 0 = 1 0 0 0 0 1
17.0> 0 1 1 0 1 = 1 0 0 0 0 1
18.0> 0 1 0 0 0 = 1 0 0 0 0 1
19.0> 0 1 0 0 1 = 1 0 0 0 0 1
20.0> 0 0 0 0 0 = 1 0 0 0 0 1
21.0> 0 0 0 0 1 = 1 0 0 0 0 1
22.0> 0 1 0 0 0 = 1 0 0 0 0 1
23.0> 0 1 0 0 1 = 1 0 0 0 0 1
24.0> 1 1 1 0 0 = 1 0 0 0 0 1
25.0> 1 1 1 0 1 = 1 0 0 0 0 1
26.0> 0 1 1 0 0 = 1 0 0 0 0 1
27.0> 0 1 1 0 1 = 1 0 0 0 0 1
28.0> 0 1 0 0 0 = 1 0 0 0 0 1
29.0> 0 1 0 0 1 = 1 0 0 0 0 1
30.0> 1 0 1 0 0 = 1 0 0 0 0 1
31.0> 1 0 1 0 1 = 1 0 0 0 0 1
32.0> 0 1 0 0 0 = 1 0 0 0 0 1
33.0> 0 1 0 0 1 = 1 0 0 0 0 1
34.0> 0 1 1 0 0 = 1 0 0 0 0 1
35.0> 0 1 1 0 1 = 1 0 0 0 0 1
36.0> 1 1 1 0 0 = 1 0 0 0 0 1
37.0> 1 1 1 0 1 = 1 0 0 0 0 1
38.0> 0 0 1 0 0 = 1 0 0 0 0 1
39.0> 0 0 1 0 1 = 1 0 0 0 0 1
40.0> 1 0 1 0 0 = 1 0 0 0 0 1
41.0> 1 0 1 0 1 = 1 0 0 0 0 1
42.0> 0 0 0 0 0 = 1 0 0 0 0 1
43.0> 0 0 0 0 1 = 1 0 0 0 0 1
44.0> 1 1 1 0 0 = 1 0 0 0 0 1
45.0> 1 1 1 0 1 = 1 0 0 0 0 1
46.0> 0 1 0 0 0 = 1 0 0 0 0 1
47.0> 0 1 0 0 1 = 1 0 0 0 0 1
48.0> 1 0 1 0 0 = 1 0 0 0 0 1
49.0> 1 0 1 0 1 = 1 0 0 0 0 1
50.0> 0 0 0 0 0 = 1 0 0 0 0 1
51.0> 0 0 0 0 1 = 1 0 0 0 0 1
52.0> 1 0 0 0 0 = 1 0 0 0 0 1
53.0> 1 0 0 0 1 = 1 0 0 0 0 1
54.0> 0 0 0 0 0 = 1 0 0 0 0 1
55.0> 0 0 0 0 1 = 1 0 0 0 0 1
56.0> 1 0 1 0 0 = 1 0 0 0 0 1
57.0> 1 0 1 0 1 = 1 0 0 0 0 1
58.0> 1 1 1 0 0 = 1 0 0 0 0 1
59.0> 1 1 1 0 1 = 1 0 0 0 0 1
60.0> 1 1 1 0 0 = 1 0 0 0 0 1
61.0> 1 1 1 0 1 = 1 0 0 0 0 1
62.0> 0 0 0 0 0 = 1 0 0 0 0 1
63.0> 0 0 0 0 1 = 1 0 0 0 0 1
64.0> 0 1 0 0 0 = 1 0 0 0 0 1
65.0> 0 1 0 0 1 = 1 0 0 0 0 1
66.0> 0 0 1 0 0 = 1 0 0 0 0 1
67.0> 0 0 1 0 1 = 1 0 0 0 0 1
68.0> 0 1 1 0 0 = 1 0 0 0 0 1
69.0> 0 1 1 0 1 = 1 0 0 0 0 1
70.0> 0 0 0 0 0 = 1 0 0 0 0 1
71.0> 0 0 0 0 1 = 1 0 0 0 0 1
72.0> 1 1 0 0 0 = 1 0 0 0 0 1
73.0> 1 1 0 0 1 = 1 0 0 0 0 1
74.0> 1 1 0 0 0 = 1 0 0 0 0 1
75.0> 1 1 0 0 1 = 1 0 0 0 0 1
76.0> 0 0 1 0 0 = 1 0 0 0 0 1
77.0> 0 0 1 0 1 = 1 0 0 0 0 1
78.0> 0 1 0 0 0 = 1 0 0 0 0 1
79.0> 0 1 0 0 1 = 1 0 0 0 0 1
80.0> 1 0 1 0 0 = 1 0 0 0 0 1
81.0> 1 0 1 0 1 = 1 0 0 0 0 1
82.0> 0 0 1 0 0 = 1 0 0 0 0 1
83.0> 0 0 1 0 1 = 1 0 0 0 0 1
84.0> 0 0 1 0 0 = 1 0 0 0 0 1
85.0> 0 0 1 0 1 = 1 0 0 0 0 1
86.0> 0 0 1 0 0 = 1 0 0 0 0 1
87.0> 0 0 1 0 1 = 1 0 0 0 0 1
88.0> 0 1 0 0 0 = 1 0 0 0 0 1
89.0> 0 1 0 0 1 = 1 0 0 0 0 1
90.0> 1 1 1 0 0 = 1 0 0 0 0 1
91.0> 1 1 1 0 1 = 1 0 0 0 0 1
92.0> 0 1 0 0 0 = 1 0 0 0 0 1
```

```
 93.0> 0 1 0 0 1 = 1 0 0 0 0 1
 94.0> 1 0 1 0 0 = 1 0 0 0 0 1
 95.0> 1 0 1 0 1 = 1 0 0 0 0 1
 96.0> 1 0 1 0 0 = 1 0 0 0 0 1
 97.0> 1 0 1 0 1 = 1 0 0 0 0 1
 98.0> 1 0 0 0 0 = 1 0 0 0 0 1
 99.0> 1 0 0 0 1 = 1 0 0 0 0 1
100.0> 1 0 0 0 0 = 1 0 0 0 0 1
101.0> 1 0 0 0 1 = 1 0 0 0 0 1
102.0> 0 1 0 0 0 = 1 0 0 0 0 1
103.0> 0 1 0 0 1 = 1 0 0 0 0 1
104.0> 0 1 0 0 0 = 1 0 0 0 0 1
105.0> 0 1 0 0 1 = 1 0 0 0 0 1
106.0> 1 0 0 0 0 = 1 0 0 0 0 1
107.0> 1 0 0 0 1 = 1 0 0 0 0 1
108.0> 0 0 1 0 0 = 1 0 0 0 0 1
109.0> 0 0 1 0 1 = 1 0 0 0 0 1
110.0> 1 1 1 0 0 = 1 0 0 0 0 1
111.0> 1 1 1 0 1 = 1 0 0 0 0 1
112.0> 1 0 0 0 0 = 1 0 0 0 0 1
113.0> 1 0 0 0 1 = 1 0 0 0 0 1
114.0> 0 1 1 0 0 = 1 0 0 0 0 1
115.0> 0 1 1 0 1 = 1 0 0 0 0 1
116.0> 0 0 0 0 0 = 1 0 0 0 0 1
117.0> 0 0 0 0 1 = 1 0 0 0 0 1
118.0> 0 1 1 0 0 = 1 0 0 0 0 1
119.0> 0 1 1 0 1 = 1 0 0 0 0 1
120.0> 1 0 1 0 0 = 1 0 0 0 0 1
121.0> 1 0 1 0 1 = 1 0 0 0 0 1
122.0> 1 1 1 0 0 = 1 0 0 0 0 1
123.0> 1 1 1 0 1 = 1 0 0 0 0 1
124.0> 1 1 1 0 0 = 1 0 0 0 0 1
125.0> 1 1 1 0 1 = 1 0 0 0 0 1
126.0> 0 0 0 0 0 = 1 0 0 0 0 1
127.0> 0 0 0 0 1 = 1 0 0 0 0 1
128.0> 1 1 0 0 0 = 1 0 0 0 0 1
129.0> 1 1 0 0 1 = 1 0 0 0 0 1
130.0> 1 0 1 0 0 = 1 0 0 0 0 1
131.0> 1 0 1 0 1 = 1 0 0 0 0 1
132.0> 1 0 0 0 0 = 1 0 0 0 0 1
133.0> 1 0 0 0 1 = 1 0 0 0 0 1
134.0> 1 1 1 0 0 = 1 0 0 0 0 1
135.0> 1 1 1 0 1 = 1 0 0 0 0 1
136.0> 0 0 0 0 0 = 1 0 0 0 0 1
137.0> 0 0 0 0 1 = 1 0 0 0 0 1
138.0> 1 0 1 0 0 = 1 0 0 0 0 1
139.0> 1 0 1 0 1 = 1 0 0 0 0 1
140.0> 1 0 0 0 0 = 1 0 0 0 0 1
141.0> 1 0 0 0 1 = 1 0 0 0 0 1
142.0> 0 1 1 0 0 = 1 0 0 0 0 1
143.0> 0 1 1 0 1 = 1 0 0 0 0 1;
```

6.3. Fault Injection and Simulation for s298 Circuit

In this section, we will execute software fault injection and simulation. It involves three basic steps:

Step 1: Input part fault injection.

- First, we will modify the original s298 vector file: Set all numbers of the first column (it represents input pin G0) to 1: it means that input G0 is stuck-at-1; then simulate s298 by using this vector file as input signal under Altera MAX PLUS II and get a non-fault-free result. Then compare the result with the result in fault-free table file and detect a stuck-at-1 fault.
- Next, we will modify the original s298 vector file: Set all numbers of the first column (it represents input pin G0) to 0: it means that input G0 is stuck-at-0; then simulate s298 by using this vector file as input signal under Altera MAX PLUS II

and get a non-fault-free result. Then compare the result with the result in fault-free table file and detect a stuck-at-0 fault.

- Repeat the same procedure from the first column to the last column. This way, simulate all input pins and test them one by one, and record all faults detected in the final fault report file.

Step 2: Inside wire fault injection.

- First, we will modify the original s298 verilog design file: Replace the first wire name with 1 in all gate-level description line; for example, "and AND3_1(G35, G10, G11, G12)" will be modified as "and AND3_1(G35, 1,G11, G12) ": this means that the first inside wire G10 is stuck-at-1; then simulate the modified s298 Verilog design file by using the original vector file as input signal under Altera MAX PLUS II and get a non-fault-free result. Then compare the result with the result in fault-free table file and detect a stuck-at-1 fault.
- Next, modify the original s298 verilog design file: Replace the first wire name with 0 in all gate-level description line; for example, "and AND3_1(G35, G10, G11, G12) " will be modified as "and AND3_1(G35, 0,G11, G12) ": this means that the first inside wire G10 is stuck-at-0; then simulate the modified s298 Verilog design file by using the original vector file as input signal under Altera MAX PLUS II and get a non-fault-free result. Then compare the result with the result in fault-free table file and detect a stuck-at-0 fault.
- Repeat the same procedure from the first inside wire to the last inside wire. This way, simulate all inside wires and test them one by one, and record all faults detected in the final fault report file.

Step 3: Output part fault injection.

- Firs, we will locate the column that represents the first output line in the fault-free output file – s298.tbl. Then in the column if we can find the value "1", that means we detected a stuck-at-0 fault for the output pin. On the other hand, if we find the value "0", it means that we detected a stuck-at-1 fault for the output pin.

- Repeat the same procedure from the first output pin to the last output pin. This way, simulate all output pins and test them one by one, and record all faults detected in the final fault report file.

6.4. Fault Detection Report on s298 Benchmark Circuit Testing

The last part of the fault simulation is concerned about generating a fault detection report. Some portions of test results on s298 benchmark circuit are furnished below.

```
% Report file for the circuit s298.bench:
 Circuit structure
   Name of the circuit                    : s298
   Number of gates                        : 119
   Number of primary inputs               : 5
   Number of primary outputs              : 6
   Number of inside wires                 :127

 ATPG parameters
   Test pattern generation mode           : RPT
   Number of random patterns              : 72
   Initial random number generator seed   : seed = rand();

 Test pattern generation results
   Number of injected faults              : 272
   Number of identified faults            : 203
   Fault simulation time                  : 660 secs

% Number of faults detected by each test pattern:

Test0 1 1 0 0 0   0 faults detected
Test1 1 1 0 0 1   0 faults detected
Test2 1 1 0 0 0   0 faults detected
Test3 1 1 0 0 1   0 faults detected
Test4 0 0 1 0 0   58 faults detected
Test5 0 0 1 0 1   0 faults detected
Test6 1 0 1 0 0   1 faults detected
Test7 1 0 1 0 1   0 faults detected
Test8 1 0 1 0 0   12 faults detected
Test9 1 0 1 0 1   0 faults detected
Test10 1 1 1 0 0   1 faults detected
Test11 1 1 1 0 1   0 faults detected
Test12 1 0 1 0 0   0 faults detected
Test13 1 0 1 0 1   0 faults detected
Test14 1 1 0 0 0   1 faults detected
Test15 1 1 0 0 1   0 faults detected
Test16 0 1 1 0 0   2 faults detected
Test17 0 1 1 0 1   0 faults detected
Test18 0 1 0 0 0   0 faults detected
Test19 0 1 0 0 1   0 faults detected
Test20 0 0 0 0 0   0 faults detected
Test21 0 0 0 0 1   0 faults detected
Test22 0 1 0 0 0   10 faults detected
Test23 0 1 0 0 1   0 faults detected
Test24 1 1 1 0 0   1 faults detected
Test25 1 1 1 0 1   0 faults detected
Test26 0 1 1 0 0   12 faults detected
Test27 0 1 1 0 1   0 faults detected
Test28 0 1 0 0 0   3 faults detected
Test29 0 1 0 0 1   0 faults detected
```

```
Test30 1 0 1 0 0   0 faults detected
Test31 1 0 1 0 1   0 faults detected
Test32 0 1 0 0 0   0 faults detected
Test33 0 1 0 0 1   0 faults detected
Test34 0 1 1 0 0   1 faults detected
Test35 0 1 1 0 1   0 faults detected
Test36 1 1 1 0 0   0 faults detected
Test37 1 1 1 0 1   0 faults detected
Test38 0 0 1 0 0   0 faults detected
Test39 0 0 1 0 1   0 faults detected
Test40 1 0 1 0 0   0 faults detected
Test41 1 0 1 0 1   0 faults detected
Test42 0 0 0 0 0   0 faults detected
Test43 0 0 0 0 1   0 faults detected
Test44 1 1 1 0 0   0 faults detected
Test45 1 1 1 0 1   0 faults detected
Test46 0 1 0 0 0   0 faults detected
Test47 0 1 0 0 1   0 faults detected
Test48 1 0 1 0 0   0 faults detected
Test49 1 0 1 0 1   0 faults detected
Test50 0 0 0 0 0   0 faults detected
Test51 0 0 0 0 1   0 faults detected
Test52 1 0 0 0 0   0 faults detected
Test53 1 0 0 0 1   0 faults detected
Test54 0 0 0 0 0   0 faults detected
Test55 0 0 0 0 1   0 faults detected
Test56 1 0 1 0 0   0 faults detected
Test57 1 0 1 0 1   0 faults detected
Test58 1 1 1 0 0   0 faults detected
Test59 1 1 1 0 1   0 faults detected
Test60 1 1 1 0 0   0 faults detected
Test61 1 1 1 0 1   0 faults detected
Test62 0 0 0 0 0   0 faults detected
Test63 0 0 0 0 1   0 faults detected
Test64 0 1 0 0 0   0 faults detected
Test65 0 1 0 0 1   0 faults detected
Test66 0 0 1 0 0   0 faults detected
Test67 0 0 1 0 1   0 faults detected
Test68 0 1 1 0 0   8 faults detected
Test69 0 1 1 0 1   0 faults detected
Test70 0 0 0 0 0   0 faults detected
Test71 0 0 0 0 1   0 faults detected
Test72 1 1 0 0 0   10 faults detected
Test73 1 1 0 0 1   0 faults detected
Test74 1 1 0 0 0   0 faults detected
Test75 1 1 0 0 1   0 faults detected
Test76 0 0 1 0 0   0 faults detected
Test77 0 0 1 0 1   0 faults detected
Test78 0 1 0 0 0   0 faults detected
Test79 0 1 0 0 1   0 faults detected
Test80 1 0 1 0 0   0 faults detected
Test81 1 0 1 0 1   0 faults detected
Test82 0 0 1 0 0   0 faults detected
Test83 0 0 1 0 1   0 faults detected
Test84 0 0 1 0 0   0 faults detected
Test85 0 0 1 0 1   0 faults detected
Test86 0 0 1 0 0   0 faults detected
Test87 0 0 1 0 1   0 faults detected
Test88 0 1 0 0 0   0 faults detected
Test89 0 1 0 0 1   0 faults detected
Test90 1 1 1 0 0   0 faults detected
Test91 1 1 1 0 1   0 faults detected
Test92 0 1 0 0 0   0 faults detected
Test93 0 1 0 0 1   0 faults detected
Test94 1 0 1 0 0   0 faults detected
Test95 1 0 1 0 1   0 faults detected
Test96 1 0 1 0 0   0 faults detected
Test97 1 0 1 0 1   0 faults detected
Test98 1 0 0 0 0   0 faults detected
Test99 1 0 0 0 1   0 faults detected
Test100 1 0 0 0 0   0 faults detected
Test101 1 0 0 0 1   0 faults detected
Test102 0 1 0 0 0   0 faults detected
Test103 0 1 0 0 1   0 faults detected
Test104 0 1 0 0 0   0 faults detected
Test105 0 1 0 0 1   0 faults detected
Test106 1 0 0 0 0   0 faults detected
Test107 1 0 0 0 1   0 faults detected
Test108 0 0 1 0 0   0 faults detected
Test109 0 0 1 0 1   0 faults detected
Test110 1 1 1 0 0   0 faults detected
Test111 1 1 1 0 1   0 faults detected
Test112 1 0 0 0 0   0 faults detected
Test113 1 0 0 0 1   0 faults detected
Test114 0 1 1 0 0   0 faults detected
Test115 0 1 1 0 1   0 faults detected
Test116 0 0 0 0 0   0 faults detected
Test117 0 0 0 0 1   0 faults detected
Test118 0 1 1 0 0   0 faults detected
Test119 0 1 1 0 1   0 faults detected
Test120 1 0 1 0 0   0 faults detected
Test121 1 0 1 0 1   0 faults detected
Test122 1 1 1 0 0   0 faults detected
Test123 1 1 1 0 1   0 faults detected
Test124 1 1 1 0 0   0 faults detected
Test125 1 1 1 0 1   0 faults detected
Test126 0 0 0 0 0   0 faults detected
Test127 0 0 0 0 1   0 faults detected
Test128 1 1 0 0 0   0 faults detected
Test129 1 1 0 0 1   0 faults detected
Test130 1 0 1 0 0   0 faults detected
Test131 1 0 1 0 1   0 faults detected
Test132 1 0 0 0 0   0 faults detected
Test133 1 0 0 0 1   0 faults detected
Test134 1 1 1 0 0   0 faults detected
Test135 1 1 1 0 1   0 faults detected
Test136 0 0 0 0 0   0 faults detected
Test137 0 0 0 0 1   0 faults detected
Test138 1 0 1 0 0   0 faults detected
Test139 1 0 1 0 1   0 faults detected
Test140 1 0 0 0 0   0 faults detected
Test141 1 0 0 0 1   0 faults detected
Test142 0 1 1 0 0   0 faults detected
Test143 0 1 1 0 1   0 faults detected
```

The fault detection report file shows the results as obtained from fault simulation. It includes the name of circuit module under test, number and name of input pins, number and name of output pins, total number of injected faults, testing run time, test vectors, and faults detected by the test vectors.

6.5. Test Results and Analysis of ISCAS 89 Sequential Benchmark Circuits

The simulation and test generation algorithms for sequential circuits were implemented in C programming language; various test sets (deterministic compacted test inputs for s27) generated for ISCAS 89 full-scan sequential benchmark circuits were used to evaluate the effectiveness of the proposed algorithms. All experimentations were carried out on an IBM-compliant Personal Computer (PC) containing Pentium III, 1000MHz, with 256 MB of RAM.

Simulations were carried out on 12 ISCAS 89 benchmark sequential circuits. Table 6.1 shows the details of these ISCAS 89 full-scan sequential benchmark circuits.

Circuit name	No. of DFFs in the circuit	No. of gates and inverters in the circuit	No. of inputs (including clock and reset) of the circuit	No. of outputs of the circuit	No. of wires of the circuit
S27	3	10	6	1	12
S298	14	119	5	6	127
S344	15	160	11	11	164
S382	21	158	5	6	173
S444	21	181	5	6	196
S526	21	193	5	6	208
S641	19	379	37	24	374
S820	5	289	20	19	275
S832	5	287	20	19	273
S953	29	395	18	23	401
S1238	18	508	16	14	512
S1423	74	657	19	5	726

Table 6.1. Basic Specifications of 12 ISCAS 89 Sequential Benchmark Circuits

Table 6.2 and Figure 6.1 show graphically the relationship among testing time, the number of DFFs, gates, inverters, and wires of ISCAS 89 sequential benchmark circuits.

Circuit name	No. of DFFs in the circuit	No. of gates and inverters in the circuit	No. of wires of the circuit	No. of test vectors used for simulation	Total testing Time (secs)
S27	3	10	12	72	60
S298	14	119	127	72	660
S344	15	160	164	186	966
S382	21	158	173	184	1200
S444	21	181	196	207	1264
S526	21	193	208	219	1249
S641	19	379	374	435	3939
S820	5	289	275	314	1917
S832	5	287	273	312	1649
S953	29	395	401	442	4102
S1238	18	508	512	542	5481
S1423	74	657	726	750	12121

Table 6.2. Testing Time and Number of DFFs, Gates, Inverters, and Wires of Benchmark Sequential Circuits

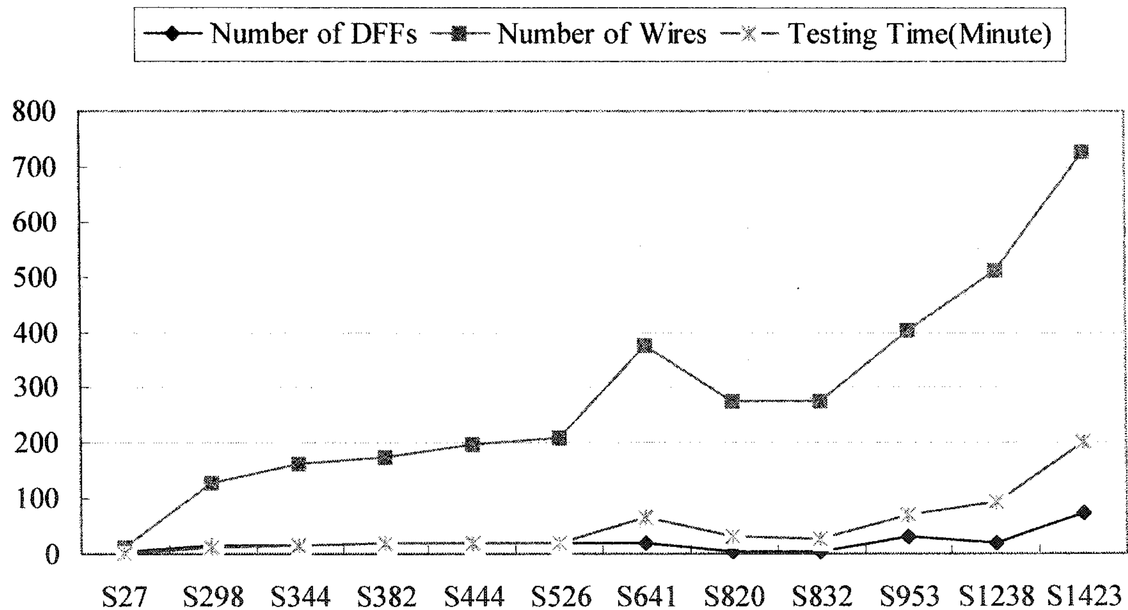


Figure 6.1. Testing time vs. no. of wires and DFFs.

From the above Table 6.2 and Figure 6.1, we can see that the testing time will increase along with increase of the number of DFFs, gates, inverters, and wires, and the number of wires plays the most important role in overall testing time.

Table 6.3 and Figure 6.2 show fault coverage with the number of DFFs, wires, and detected faults of the tested ISCAS 89 benchmark circuits.

Circuit name	No. of DFFs in the circuit	No. of wires in the circuit	No. of detected faults	Fault coverage (%)
S27	3	12	34	100
S298	14	127	203	74.6
S344	15	164	343	93.2
S382	21	173	90	24.7
S444	21	196	86	21.0
S526	21	208	79	18.2
S641	19	374	758	87.5
S820	5	275	317	50.8
S832	5	273	345	55.6
S953	29	401	834	94.8
S1238	18	512	885	82.0
S1423	74	726	581	38.8

Table 6.3. Fault Coverage, Number of DFFs, Wires and Detected Faults

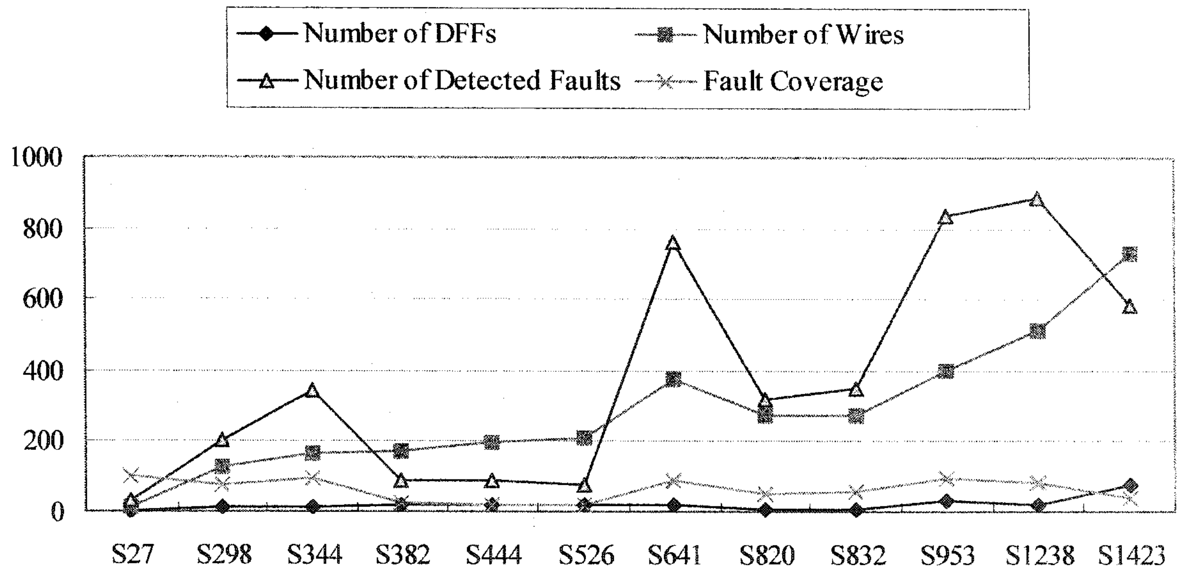


Figure 6.2. Fault coverage vs. no. of DFFs and wires.

Table 6.4 and Figure 6.3 show the number of injected faults, test vectors and detected faults of all tested ISCAS 89 circuits.

Circuit name	No. of faults injected	No. of test vectors used for simulation	No. of faults detected
S27	34	72	34
S298	272	72	203
S344	368	186	343
S382	364	184	90
S444	410	207	86
S526	434	219	79
S641	866	435	758
S820	624	314	317
S832	620	312	345
S953	880	442	834
S1238	1080	542	885
S1423	1496	750	581

Table 6.4. Number of Injected Faults, Test Vectors, and Detected Faults

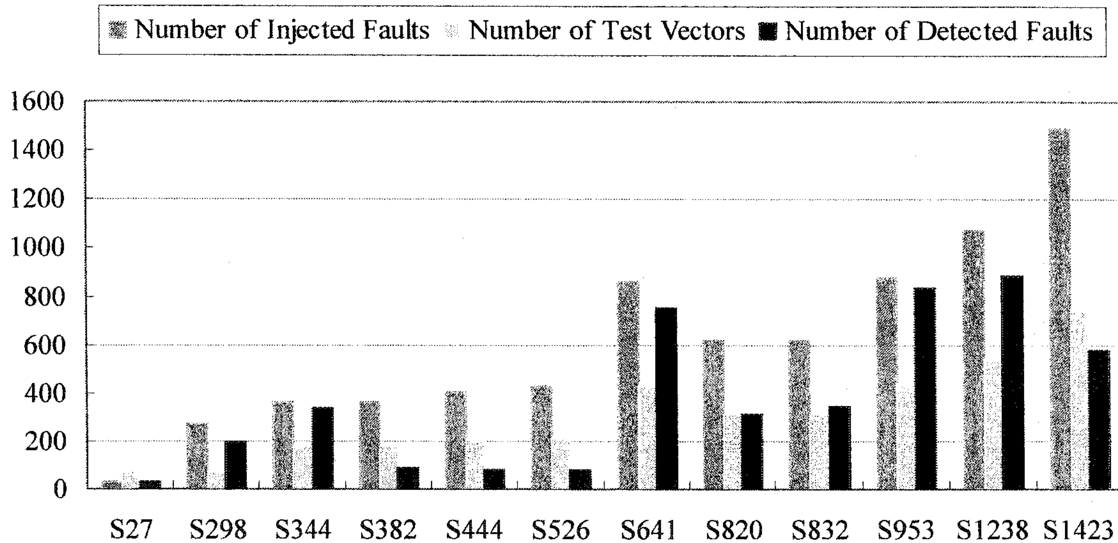


Figure 6.3. Detected faults vs. injected faults, test vectors.

Many factors affect the circuit module fault coverage. Theoretically, we can get almost 100% fault coverage using exhaustive test vectors for sequential circuits. Table 6.5 shows test results and parameters of all 12 ISCAS 89 sequential benchmark circuits.

MUT name	No. of DFFs in the MUT	No. of Gates and Inverter-s in the MUT	No. of inputs (including clock and reset) of the MUT	No. of outputs of the MUT	No. of wires of the MUT	No. of test vectors used for testing	Total testing time (secs)	No. of faults injected	No. of faults detected	Fault cove-rage (%)
S27	3	10	6	1	12	72	60	34	34	100
S298	14	119	5	6	127	72	660	272	203	74.6
S344	15	160	11	11	164	186	966	368	343	93.2
S382	21	158	5	6	173	184	1200	364	90	24.7
S444	21	181	5	6	196	207	1264	410	86	21.0
S526	21	193	5	6	208	219	1249	434	79	18.2
S641	19	379	37	24	374	435	3939	866	758	87.5
S820	5	289	20	19	275	314	1917	624	317	50.8
S832	5	287	20	19	273	312	1649	620	345	55.6
S953	29	395	18	23	401	442	4102	880	834	94.8
S1238	18	508	16	14	512	542	5481	1080	885	82.0
S1423	74	657	19	5	726	750	12121	1496	581	38.8

Table 6.5. Simulation Results on ISCAS 89 Sequential Benchmark Circuits

Chapter 7
CONCLUSIONS

A Verilog HDL-based fault simulator for testing embedded cores-based synchronous sequential circuits is proposed in the thesis to detect single stuck-line faults. The simulator emulates a typical BIST (built-in self-testing) environment with test pattern generator that sends its outputs to an MUT (module under test) and the output streams from the MUT are fed into a response data analyzer. The response data analyzer is comprised of a storage unit for the fault-free responses from the MUT and a comparator. A fault is detected if the circuit response is different from that of the fault-free circuit. All of the faults detected are recorded in a detailed simulation results file. The fault simulator is suitable for testing sequential circuits described in Verilog HDL. The automatic fault simulator generates tests for the MUT described at the gate and flip-flop level. There are very few constraints on the circuit. The simulation process is completely automatic, and requires no intervention from the designer during the test generation process. The thesis describes in detail the architecture and applications of the fault simulator along with the models of sequential elements used. Results on some simulation experiments on ISCAS 89 sequential benchmark circuits are also provided.

It is hoped that this research will lead to further work in this domain. The future research can be classified into these categories: improvements of the fault simulator by augmenting its enhancements and performance, extending the test method to handle several MUTs (module under test) at same time and developing new integrated test wrapper for IP (intellectual property) cores.

In the context of improving the performance of the fault simulator, several things might be considered:

- It is possible to preprocess the circuit to identify those primary inputs that do not influence the detection of the faults under consideration. During the search for a test for those faults, such primary inputs could then remain unchanged.
- Currently, the input signals are modified one at a time. This strategy is simple and works in most cases. However, one can develop more sophisticated strategy for generating trial vectors and use a more sophisticated mechanism in subsequent

passes. In this way, a certain percentage of fault coverage can be obtained within a reasonable computing time, and higher fault coverage can still be reached if longer computing time is affordable.

- The proposed approach can also be applied for those sequential circuits described by VHDL or AHDL after making slight change in the program.

REFERENCES

[1] S.R.Das, "Random Test Generation for LSI/VLSI Circuits – a Survey", IEEE VLSI Technical Bulletin, pp. 4-11, March 1987.

[2] S.R.Das, Z.Chen, S.M.Wu, S.Y.Lee and A.Bhattacharyya, "On the Design of Improved Failure Detection Experiments in Synchronous Sequential Machines Based on Terminal Measurements", Computers and Electrical Engineering, pp. 293-297, December 1979.

[3] S.R.Das, "Nontransient Fault Testing in Sequential Logic Circuits using Stochastic Modeling and Simulation", Cybernetics and Systems: An International Journal, pp. 15-27, January 1989.

[4] P.H.Bardell, W.H.McAnney, and J.Savir, "Built-In Test for VLSI: Pseudorandom Techniques", Wiley Interscience, New York, NY, 1987.

[5] Neil H.E.Weste, Kamran Eshraghian, "Principles of CMOS VLSI Design: A Systems Perspective (Second Edition)", ADDISON-WESLEY Longman Publishing Co., 1993.

[6] O. H. Ibarra and S. K. Sahni, "Polynomially complete fault detection problems", IEEE Trans. Comp., Vol. C-24, pp. 242-249, March 1975.

[7] H. Fujiwara and S. Toida, "The complexity of fault detection problems for combinational logic circuits", IEEE Trans. Comp., Vol. C-31, pp.555-560, June 1982.

[8] E.G.Ulrich, "Exclusive Simulation of Activity in Digital Networks", Communications of the ACM, Vol. 12, pp. 102-110, February 1969.

[9] R.D. Eldred, "Test Routines Based on Symbolic Logic Statements", J. ACM, Vol. 6, No. 1, pp.33-36, Jan. 1959.

[10] J.F.Poage, "Derivation of Optimum Tests to detect Faults in Combinational Circuits", pp. 483-528 in Proc. Symp. On Mathematical Theory of Automata (April 1962), Polytechnic Press, New York, 1963.

[11] C.Y.Lo, H.N.Nham and A.K.Bose, "Algorithms for an Advanced Fault Simulation System in MOTIS", IEEE Trans. CAD, Vol. CAD-6, pp.232-240, March 1987.

[12] P.Goel, "The Feed Forward Logic Model in the Testing of Large Scale Integrated Logic Circuits", Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, PA, September 1973.

[13] C.W.Cha, "Multiple Fault Diagnosis in Combinational Networks", Proc. 16th, Des. Auto. Conf., San Diego, CA, pp.149-155, June 1979.

[14] V.D. Agrawal and S.C. Seth, Tutorial – Test Generation for VLSI Chips, IEEE Computer Society Press, Washington, D.C., 1988.

[15] E.W.Thompson and S.A. Szygenda, "Digital Logic Simulation in a Time-Based, Table-Driven Environment, Part 2, Parallel Fault Simulation", Computer, Vol. 8, pp. 38-44, March 1975.

[16] D.B.Armstrong, "A Deductive Method for Simulating Faults in Logic Circuits", IEEE Trans. Computers, Vol. C-21, pp.464-471, May 1972.

[17] E.G.Ulrich and T.Baker, "The Concurrent Simulation of Nearly Identical Digital Networks", Computer, pp. 39-44, April 1974.

[18] M.D. Schuster and R.E. Bryant, "Concurrent Fault Simulation of MOS Digital Circuits", Proc. Conf. Adv. Res. In VLS*I*, Cambridge, MA, pp. 129-138, January 1984.

[19] T.Sridhar and J.P.Hayes, "A Functional Approach to Testing Bit-Sliced Microprocessors", IEEE Trans. Comp., Vol. C-30,pp.563-571, Aug.1981.

[20] W.A Rogers and J..A..Abraham, "CHIEFS: A Concurrent, Hierarchical and Extensible Fault Simulator", Proc. Int. Test Conf., Philadelphia, PA, pp. 710-716, Nov. 1985.

[21] T.Blank, "A Survey of Hardware Accelerators Used in Computer-Aided Design", IEEE Design & Test of Computers, pp.21-39, August, 1984.

[22] P. Goel, "Test Generation Costs Analysis and Projections", Proc. 17th Design Automation Conference, Minneapolis, MN, pp.77-84, June 1980.

[23] D.Brahme and J.A.Abraham, "Functional Testing of Microprocessors", IEEE Trans. Comp., Vol. C-33, pp.475-485, June 1984.

[24] V.D. Agrawal, "Sampling Techniques for Determining Fault Coverage in LSI Circuits", Journal of Digital Systems, Vol.5, pp.189-202, 1981.

[25] Y.H.Levendel and P.R.Menon, "Fault Simulation Methods – Extensions and Comparison", Bell Sys. Tech. Jour., Vol. 60, pp. 2235-2258, November 1981.

[26] V.D.Agrawal, S.C.Seth, and P.Agrawal, "Fault Coverage Requirement in Production Testing of LSI Circuits", IEEE Journal of Solid-State Circuits, Vol. SC-27, pp.57-61, Feb. 1982.

[27] S.C.Seth and V.D.Agrawal, "Characterizing the LSI Yield from Wafer Test Data", IEEE Trans. on CAD, Vol. CAD-3, pp.123-126, April 1984.

[28] T.W.Williams and N.C.Brown, "Defect Level as a Function of Fault Coverage", IEEE Trans. Computers, Vol. C-30, pp. 987-988, December 1981.

[29] J.P.Roth, W.G.Bouricius, and P.R.Schneider, "Programmed Algorithms to Compute Tests and to Detect and Distinguish Between Failures in Logic Circuits", IEEE Trans. on Electronic Computers, Vol.EC-16, pp.567-580, October 1967.

[30] P.Muth, "A Nine-Valued Circuit Model for Test Generation", IEEE Trans.Comp., Vol. C-25, pp.630-636, June 1976.

[31] P.Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits", IEEE Trans. Comp., Vol. C-30, pp.215-222, March 1981.

[32] T.J. Snethen, "Simulator Oriented Fault Test Generator", Proc. 14th Des. Auto. Conf., New Orleans, Louisianan, pp. 88-93, June 1997.

[33] E. Kjelkerud and O. Thessen, "Generation of Hazard Free Tests using the D-Algorithm in a Timing Accurate System for Logic and Deductive Fault Simulation", Proc. Des. Auto. Conf., San Diego, CA, pp. 180-184, June 1979.

[34] R.A.Marlett, "EBT: A Comprehensive Test Generation Technique for Highly Sequential Circuits", Proc. Des. Auto. Conf., Las Vegas, Nevada, pp. 335-339, June 1978.

[35] W.T.Cheng, "The BACK Algorithm for Sequential Test Generation", Proc. Int. Conf. Computer Design (ICCD-88), Rye Brook, NY, pp. 66-69, October 1988.

[36] H.-K. T. Ma, S. Devadas, A.R. Newton, and A. Sangiovanni-Vincentelli, "Test Generation for Sequential Finite State Machines", Int, Conf. Computer-Aided Design (ICCAD '87), pp. 288-291, Nov. 1987.

[37] Donald E. Thomas, Philip R.Moofby, "The VERILOG Hardware Description Language", Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[38] M.H.Assaf, Space Compactor Design for Built-In Self-Testing of VLSI Circuits from Compact Test Sets using Sequence Characterization and Failure Probabilities. M.A.Sc. Thesis, Department of Electrical Engineering, University of Ottawa, Ottawa, Ontario, August 1996.

[39] J.Y.Liang. Response Data Compaction in BIST under Generalized Mergeability Based on Switching Theory Formulation and utilizing a New Measure of Failure Probability. M.A.Sc.thesis, School of Information Technology and Engineering, University of Ottawa, Ottawa, Ontario, September 2000.

[40] F.Brglez, D.Bryan and K.Kozminski, "Combinational Profiles of Sequential Benchmark Circuits", Proc. Int. Symp. on Circuits and Systems, pp. 1929-1934, May 1989.

[41] F.J.Hill and B.Huey, "A Design Language Based Approach to Test Sequence Generation", Computer, Vol.10, pp.28-33, June 1977.

[42] M.J.Bending, "Hitest: A Knowledge-Based Test Generation System", IEEE Design & Test of Computers, Vol.1, pp.83-92, May 1984.

[43] V.D.Agrawal and S.M.Reddy, "Fault Modeling and Test Generation", in VLSI Handbook, J.DiGiacomo, Ed., McGraw-Hill, New Youk, 1989.

[44] S.R.Das, C.V.Ramamoorthy, M.H.Assaf, E.M.Petriu and W.B.Jone, "Fault tolerance in systems design in VLSI using data compression under constraints of failure probabilities", IEEE Trans. Instr.Meas., pp.1725-1747, December 2001.

[45] H.K.Lee and D.S.Ha, "HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits, Proc. Design Automation Conf., pp.336-340, June 1992.

[46] S.R.Das and A.Bhattacharyya, "Fault Detection in Sequential Machines with Increased Fault Coverage", Electronics Letters, pp.28-29, January 1978.

[47] C.Bellon, C.Tobach, and G.Saucier, "VLSI Test Program Generation: A System for Intelligent Assistance", Proc.Int. Conf. Comp.Des.(ICCD'83), Port Chester, NY, pp..49-52, October 1983.

[48] http://www.fm.vslib.cz/~kes/asic/iscas/

[49] http://www.visc.vt.edu/~rlajauni/benchmarks/ISCAS89/verilog/

[50] R.David and P.Thevenod-Fosse, "Random Testing of Integrated Circuits", IEEE Transactions on Computers, pp. 20-25, March 1981.

[51] J.A.Waicukauski, E.B.Eichelberger, D.O.Forlenza, E.Lindbloom, and T.McCarthy, "Fault simulation for structured VLSI", VLSI Syst.Des., vol.6,no.12,pp 20-32,Dec.1985.

[52] Felice Balarin, "Verilog HDL Modelling styles for formal verification", D.Agnew and L. Claesen, editors, Computer Hardware Description Languages and their Applications (A-32), pp.453-466, Elsevier Science Publishers BV (North-Holland), 1993

[53] http://www.sutherland-hdl.com/on-line_ref_guide/vlog_ref_top.html/

[54] http://www.verilog.com/

[55] IEEE Standard Hardware Description Language Based on the Verilog Hardware description Language, IEEE Std 1364-1995.14 October 1996.

[56] Verilog Hardware Description Language Reference Manual, Version 2.0, March 1993, Open Verilog International, San Jose, CA.

[57] Altera Corporation, Data Book, San Jose, CA, 1995.

[58] MAX+PLUS II Users Guide, Altera Inc., San Jose, CA, 1992.

[59] http://www.altera.com/support/software/sof-maxplus2.html/

[60] J.Losq, "Efficiency of Random Compact Testing", IEEE Transactions on Computers, pp. 516-525, June 1978.

[61] D.H.Lee and S.M.Reddy, "On efficient fault simulation for synchronous sequential circuits", Design automation Conf., pp. 327-331, June 1992.

[62] P.K. Lala, "Fault Tolerant and Fault Testable Hardware Design", Prentice-Hall International Inc., London, 1985.

[63] Kwang-Ting Cheng and Vishwani D.Agrawal, "Unified Methods for VLSI Simulation and Test Generation",Kluwer Academic Publishers,1989.

[64] D.P.Siewiorek and L.K.W.Lai, "Testing of Digital Systems", IEEE Transactions on Computers, pp.1321-1333, October 1981.

[65] N.Singh, "An Artificial Intelligence Approach to Test Generation", Kluwer Academic Publishers, Norwell, MA, 1987.

[66] M.Brashler, D.Coleman, and R.Dubois, "An Integrated IC Test Development System", Proc. Custom Integrated Circuits Conf., Rochester, NY, pp. 169-171, May 1984.

[67] S.R.Das and W.B.Jone, "Modified Transiton Matrix and Fault Testing in Sequential Logic Circuits under Random Stimuli with a Specified Measure of Confidence", Cybernetics and Systems: An International Journal, pp.1-12, 1986.

LIST OF PAPERS BY THE CANDIDATE

1. Chuan Jin, "Hardware and Software CO-Design in Space Compaction of Cores-Based Digital Circuits", Instrumentation and Measurement Technology Conference (IMTC 2004), Como, Italy, 18-20 May 2004.
   (with Sunil R. Das, Emil M. Petriu, Voicu Groza and Mansour H. Assaf)

2. Chuan Jin, "Implementation of a Testing Environment for Digital IP Cores", Instrumentation and Measurement Technology Conference (IMTC 2004), Como, Italy, 18-20 May 2004.
   (with Sunil R. Das, Emil M. Petriu, Mansour H. Assaf and Wen-Ben Jone)

3. Chuan Jin, "Testing Embedded Cores-Based System-On-A-Chip (SOC) – Test Architecture and Implementation", Modeling, Identification and Control Conference (MIC 2004), February 23-25, 2004, Grindelwald, Switzerland.
   (with Sunil R. Das, Emil M. Petriu, Mansour H. Assaf and Liwu Jin)

4. Chuan Jin, "Test Implementation of Embedded Cores-based Sequential Circuits Based on Verilog HDL Under ALTERA MAX PLUS II Development Environment", Integrated Design & Process Technology (IDPT) Conference, June 28 - July 2, 2004, Kusadasi, Turkey.
   (with Sunil R. Das, Emil M. Petriu, Mansour H. Assaf and Mehmet Sahinoglu)

```
//*****************************************************
//      Define Global Variables Used in SqTesting_1.
//*****************************************************


#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <dos.h>
#define   MAX   2048


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

unsigned long int random_number(unsigned long int seed)
{
   const unsigned long int module = 2147483647L;
   const unsigned long int multiplier = 16807L;
   static unsigned long int ran = seed;

   // initial value is used only at the first time

   ran = (ran * multiplier) % module;
   return ran;
}

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



//*******************************************************
//                    Main Function
//*******************************************************


int main( )
{
   struct  time t;
   int t1hour,t1minute,t1second;
   int t2hour,t2minute,t2second;
   unsigned long int seed;
     FILE *Dtem1;
     FILE *Dtem2;
     char DaLine[MAX];
     char Dline1[MAX],Dline2[MAX];
     char *Dptr,*ptr1,*Dptr2;
     int Dk,Dl;
     int n, m, gen;
     int i,mi,wireline;
  char MUTname[10],MUT[10];
  char
vec[20],tbl[20],vecbak[20],vectemp[20],tbak[20],rpt[20],MUTx[20],vlgbak[20];
  char cpvecbak[40], cptbl[40],cml[80],cmlx[80],sim[80],simx[80],delv[40];
  char cpvt[40],delvt[40],cpvlg[40],cpvec[40],cpv[40],cpx[40],delspeed[40];
  char w0[20],w1[20],w2[20];
        // char *vec;
        //char *tbl,*vbak,*tbak;
        int flg1=0,flg2=0;
        char inc[800],outc[800],wirec[1800];
        // char *inc;
```

```
    // char *outc;
    // char incbak[40];
    int innum,outnum,wirenum;

char ch[800][1000] = {"\0"};
FILE *tem1;
FILE *tem2;
FILE *input;
FILE *output;
FILE *vtbak;
FILE *tblbak;
FILE *report;
char aLine[MAX];
char line2[MAX];

char toFindws[]= "//# wires";
char toFindins[]= "//# inputs";
char toFindouts[]= "//# outputs";
char toFind1[] = "wire ";
char toFind2[] = "input ";
char toFind3[] = "output ";
char toFind4[] = ".0>";
char *ptr,*ptr1,*ptr2;
char a[1024],aa[1024];
char b[1024];
char c[1024];
char ijf[2],pinname[4];
char itostr[6];

int com, testno=0;
int count[100],countfault[1000];
int j,r=1,s=0;
int r1=1,s1=0;
int pin,pat=1,wir=0;
int bz=-1,ccc=0;
int findfault,findfaultnum=0;
int loopnum=0;

    printf ("Enter Name of MUT: ");

    cin >> MUTname;

    strcpy(MUT,MUTname);
    strcpy(vec,MUTname);
    strcpy(tbl,MUTname);
    strcpy(vecbak,MUTname);
    strcpy(vectemp,MUTname);
    strcpy(tbak,MUTname);
    strcpy(rpt,MUTname);
    strcpy(vlgbak,MUTname);
    strcpy(MUTx,MUTname);

    strcat(MUTname,".v");
    strcat(vec,".vec");
    strcat(tbl,".tbl");
    strcat(vecbak,"vec.bak");
    strcat(vectemp,".bak");
    strcat(tbak,"tbl.bak");
    strcat(rpt,"report.log");
    strcat(vlgbak,"v.bak");
    strcat(MUTx,"x.v");

    printf("MUT is %s\n",MUT);
```

```
   printf("MUTname is %s\n",MUTname);
   printf("vec is %s\n",vec);
   printf("vecbak is %s\n",vecbak);
   printf("vectemp is %s\n",vectemp);
   printf("tbak is %s\n",tbak);
   printf("rpt is %s\n",rpt);
   printf("vlgbak is %s\n",vlgbak);
   printf("MUTx is %s\n",MUTx);

//------------------------------------------------------------
           strcpy(cpvecbak,"copy ");
           strcat(cpvecbak,vec);
           strcat(cpvecbak," ");
           strcat(cpvecbak,MUT);
           strcat(cpvecbak,"vec.bak");
           printf("cpvecbak is %s\n",cpvecbak);

           strcpy(cpvec,"copy ");
           strcat(cpvec,vecbak);
           strcat(cpvec," ");
           strcat(cpvec,MUT);
           strcat(cpvec,".vec");
           printf("cpvec is %s\n",cpvec);

           strcpy(cml,"c:\\maxplus2\\maxplus2 -c ");
           strcat(cml,MUT);
           //strcat(cml," -s -vec d:\\a\\");
           //strcat(cml,vec);
           //strcat(cml," -tbl d:\\a\\");
           //strcat(cml,tbl);
           //strcat(cml," d:\\a\\");
           //strcat(cml,MUT);
           printf("cml is %s\n",cml);

           strcpy(cmlx,"c:\\maxplus2\\maxplus2 -c ");
           strcat(cmlx,MUTx);
           //strcat(cml," -s -vec d:\\a\\");
           //strcat(cml,vec);
           //strcat(cml," -tbl d:\\a\\");
           //strcat(cml,tbl);
           //strcat(cml," d:\\a\\");
           //strcat(cml,MUT);
           printf("cmlx is %s\n",cmlx);

           strcpy(sim,"c:\\maxplus2\\maxplus2 -s");
           strcat(sim," -vec ");
           strcat(sim,vec);
           strcat(sim," -tbl ");
           strcat(sim,tbl);
           strcat(sim," ");
           strcat(sim,MUT);
           printf("sim is %s\n",sim);

           strcpy(simx,"c:\\maxplus2\\maxplus2 -s");
           strcat(simx," -vec ");
           strcat(simx,vec);
           strcat(simx," -tbl ");
           strcat(simx,tbl);
           strcat(simx," ");
           strcat(simx,MUTx);
           printf("simx is %s\n",simx);

           strcpy(cptbl,"copy ");
```

```
            strcat(cptbl,tbl);
            strcat(cptbl," ");
            strcat(cptbl,MUT);
            strcat(cptbl,"tbl.bak");
            printf("cptbl is %s\n",cptbl);

            strcpy(delv,"del ");
            strcat(delv,vec);
            printf("delv is %s\n",delv);

            strcpy(cpvt,"copy ");
            strcat(cpvt,vectemp);
            strcat(cpvt," ");
            strcat(cpvt,vec);
            printf("cpvt is %s\n",cpvt);

            strcpy(delvt,"del ");
            strcat(delvt,vectemp);
            printf("delvt is %s\n",delv);

            strcpy(cpvlg,"copy ");
            strcat(cpvlg,MUTname);
            strcat(cpvlg," ");
            strcat(cpvlg,vlgbak);
            printf("cpvlg is %s\n",cpvlg);

            strcpy(cpx,"copy ");
            strcat(cpx,MUTname);
            strcat(cpx," ");
            strcat(cpx,MUTx);
            printf("cpx is %s\n",cpx);

            strcpy(cpv,"copy ");
            strcat(cpv,vlgbak);
            strcat(cpv," ");
            strcat(cpv,MUTname);
            printf("cpv is %s\n",cpv);

            strcpy(delspeed,"del ");
            strcat(delspeed,MUT);
            strcat(delspeed,".CNF");

            system(cpvlg);     //copy .v file to v.bak

//---------------------------------------------------------------------------
//                 To get input_number of MUT
//---------------------------------------------------------------------------

      input = fopen(vlgbak, "r");
      while ( fgets(aLine, MAX, input) != NULL )
      {
        if ( (strstr(aLine,toFindins)) != NULL ) //To find the key word
"inputs".
          {
            ptr = strstr(aLine,toFindins);
            ptr=ptr+11;
            for (int i=0;i<8;++i)
              {
                if (*ptr !='\0')
                 {
                  inc[i]=*ptr;
                  ++ptr;
                 }
```

```
              else
               {
                 inc[i]='\0';
               }
             }
         }
//-----------------------------------------------------------------------------
//                  To get output_number of MUT
//-----------------------------------------------------------------------------

      if ( (strstr(aLine,toFindouts)) != NULL ) //To find the key word
"outputs".
           {
              ptr = strstr(aLine,toFindouts);
              ptr=ptr+12;
              for (int i=0;i<8;++i)
              {
                if (*ptr !='\0')
                 {
                  outc[i]=*ptr;
                  ++ptr;
                 }
                else
                 {
                  outc[i]='\0';
                 }
               }
          }
//-----------------------------------------------------------------------------
//                  To get wire_number of MUT
//-----------------------------------------------------------------------------

         if ( (strstr(aLine,toFindws)) != NULL )  //To find the key word
"wires".
              {
               ptr = strstr(aLine,toFindws);
               ptr=ptr+10;
               for (int i=0;i<8;++i)
                {
                  if (*ptr !='\0')
                   {
                    wirec[i]=*ptr;
                    ++ptr;
                   }
                  else
                   {
                    wirec[i]='\0';
                   }
                }
              }
          }


      fclose(input);
      innum=atoi(inc);
      outnum=atoi(outc);
      wirenum=atoi(wirec);
      printf("wirec is %s\n",wirec);
      printf("wirenum is %d\n",wirenum);
      printf("inc is %s\n",inc);
      printf("outc is %s\n",outc);
      printf("innum is %d\n",innum);
```

```
        printf("outnum is %d\n",outnum);


//**********************************************************
//   Function for getting input_name of MUT
//**********************************************************

       input = fopen(vlgbak, "r");
       while ( fgets(aLine, MAX, input) != NULL )
       {
         if ( (strstr(aLine,toFind2)) != NULL )
          {
            ptr = strstr(aLine,toFind2);
            ptr=ptr+6;
               for (int i=0;i<800;++i)
                {
                  if (*ptr !=';')
                   {
                    inc[i]=*ptr;
                    ++ptr;
                   }
                  else
                   {
                     inc[i]='\0';
                     break;
                   }
                }
           }
        }
         fclose(input);
         i=0;
          while (inc[i]!='\0')
          {
            if (inc[i]==',')
             {
                inc[i]=' ';
              }
               ++i;
          }
        printf("inc is %s\n",inc);


//**********************************************************
//   Function for getting output_name of MUT
//**********************************************************

       input = fopen(vlgbak, "r");
       while ( fgets(aLine, MAX, input) != NULL )
       {
         if ( (strstr(aLine,toFind3)) != NULL )
          {
            ptr = strstr(aLine,toFind3);
            ptr=ptr+7;
               for (int i=0;i<800;++i)
                {
                  if (*ptr !=';')
                   {
                    outc[i]=*ptr;
                    ++ptr;
                   }
                  else
                   {
```

```
                    outc[i]='\0';
                    break;
                  }
               }
            }
         }
         fclose(input);
         i=0;
         while (outc[i]!='\0')
         {
            if (outc[i]==',')
             {
              outc[i]=' ';
             }
              ++i;
          }
        printf("outc is %s\n",outc);

//************************************************************
//   Function for getting wire_name of MUT
//************************************************************

      input = fopen(vlgbak, "r");
      while ( fgets(aLine, MAX, input) != NULL )
       {
         if ( (strstr(aLine,toFind1)) != NULL )
          {
            ptr = strstr(aLine,toFind1);
            ptr=ptr+5;
               for (int i=0;i<1800;++i)

                {
                  if (*ptr !=';')
                   {
                    wirec[i]=*ptr;
                    ++ptr;
                   }
                  else
                   {
                     wirec[i]=',';
                     wirec[i+1]='\0';
                     break;
                   }
                 }
            }
         }
         fclose(input);
         printf("wirec is %s\n",wirec);

//****************************************************************************
//             Function for deterministic input test patterns
//****************************************************************************

/*      Dtem1 = fopen("vold.txt", "r");
        Dtem2 = fopen("vnew.txt","a");
        Dk=0;
        while ( fgets(DaLine, MAX, Dtem1) != NULL )
         {
                if ( (Dptr1 = strstr(DaLine,">")) != NULL )
                {
                   strcpy(DaLine, Dptr1 + 2);
```

```
                strncpy(Dline1, DaLine, 8);
                Dline1[8] = '\0';
                strncpy(Dline2, DaLine, 8);
                Dline2[8] = '\0';

                  fprintf (Dtem2, "%d > ",Dk);
                  strcat(Dline1,"0 0");
                  fprintf (Dtem2, "%s",Dline1);
                  fprintf (Dtem2, "\n");
                  Dk=Dk+1;

                  fprintf (Dtem2, "%d > ",Dk);
                  strcat(Dline2,"0 1");
                  fprintf (Dtem2, "%s",Dline2);
                  fprintf (Dtem2, "\n");
                  Dk=Dk+1;
                }
        }

                    fclose(Dtem2);
                    fclose(Dtem1);          */


//*****************************************************************************
//             Function for generating random vectors file
//*****************************************************************************

      srand(time(NULL));
      seed = rand();
      n=innum;
            m=2*(innum+outnum+wirenum);
      printf ("Random numbers are: \n");
             input = fopen(vec, "w");
             fprintf (input, "START 0 ;\n");
             fprintf (input, "INTERVAL 1 ;\n");
             fprintf (input, "INPUTS ");
             fprintf (input, inc);
             fprintf (input, ";\n");
             fprintf (input, "rst CK ;\n");
             fprintf (input, "PATTERN\n");

        int k=0;
        for (int i = 0; i < m; ++i)
        {
            for (int count = 0; count < 2*(n-2); ++count)
               { gen = random_number(seed) % 2;
                if (gen == 0) ch[i][count] = '0';
                else ch[i][count] = '1';
                  ++count;
                  ch[i][count]=' ';
                }
                printf ("%s\n", ch[i]);

                if (k < m)          // To add rst & CK to vector file
                {
                 fprintf (input, "%d> %s",k,ch[i]);
                   if (k<1)
                    {
                     fprintf (input, "1 ");
                    }
                   else
                    {
                     fprintf (input, "0 ");
```

```
                    }
                    gen = k % 2;
                     if (gen == 0)
                     {
                       fprintf (input, "1 \n");
                     }
                     else
                     {
                      fprintf (input, "0 \n");
                     }
                      k=k+1;
                      gen=k%2;
                      if (gen==0)
                      {
                               fprintf (input, "%d> %s",k,ch[i]);
                                if (k<1)
                                 {
                                  fprintf (input, "1 ");
                                 }
                                else
                                 {
                                  fprintf (input, "0 ");
                                 }
                                  gen = k % 2;
                                  if (gen == 0)
                                  {
                                   fprintf (input, "1 \n");
                                  }
                                  else
                                  {
                                    fprintf (input, "0 \n");
                                  }
                                  k=k+1;
                        }
                    }
        }
                fprintf (input, ";\n");
                fprintf (input, "OUTPUTS ");
                fprintf (input, outc);
                fprintf (input, ";\n");
                fclose(input);
// End of forming random vector file.

       system(cpvecbak);    // Copy vector file to vector backup file
       system(delspeed);
       system(cml);
       system(sim);
       system(cptbl);

       gettime(&t);         // Record the start time
       t1hour=t.ti_hour;
       t1minute=t.ti_min;
       t1second=t.ti_sec;


//*************************************************************************
//                Function for input-fault-injection
//*************************************************************************

     bz=-1;
     for (pin=1;pin<(n-1);pin++)
            {
                if (bz<0)
```

```
            {
             strcpy(ijf,"0");
            }
          else
           {
            strcpy(ijf,"1");
           }
           printf("ijf is %s\n",ijf);

            tem1 = fopen(vecbak, "r");
            tem2 = fopen(vec,"w");
      while ( fgets(aLine, MAX, tem1) != NULL )
       {
          if ( (ptr = strstr(aLine,"> ")) != NULL )
            {
               ptr = ptr + (2*pin);
               strcpy(line2, ptr + 1);
               *ptr = 0;
               strcat(aLine, ijf);
               strcat(aLine, line2);
             }
              fprintf(tem2, "%s", aLine);
       }
    fclose(tem1);
    rewind(tem2);
    fclose(tem2);

    system(delspeed);
    system(cml);
    system(sim);
    loopnum=loopnum+1;


//************************************************************************
//                Function for comparison of 2 result table files
//                         for input-fault-injection
//************************************************************************

    tem1 = fopen(tbl, "r");
    tem2 = fopen(tbak,"r");
    r=-1;
    s=-1;
    findfault=0;
    while ( fgets(aLine, MAX, tem1) != NULL )
     {
           if ( (ptr1 = strstr(aLine,"0>")) != NULL )
           {
              r=r+1;
              strcpy(aLine, ptr1 + 3);

               while ( fgets(line2, MAX, tem2) != NULL )
               {
                if ( (ptr2 = strstr(line2,"0>")) != NULL )
                 {
                   s=s+1;
                   if (s==r)
                    {
                      strcpy(line2, ptr2 + 3);
                      if ((s>0)&&(strcmp(aLine,line2)!=0))
                       {
                         countfault[r]=countfault[r]+1;
                         itoa(pin,itostr,10);
                         strcpy(cpvt,"copy ");
```

```
                              strcat(cpvt,tbl);
                              strcat(cpvt," ");
                              strcat(cpvt,MUT);
                              strcat(cpvt,itostr);
                              strcat(cpvt,ijf);
                              strcat(cpvt,"tbl.bak");
                              printf("cpvt is %s\n",cpvt);
                              system(cpvt);
                              findfaultnum=findfaultnum+1;
                              findfault=1;
                              break;
                            }
                            else
                            {  break; }
                         }
                       }
                      }
                    if (findfault==1)
                     {break;}
                  }
           }
           fclose(tem1);
           fclose(tem2);

//-------------------------------------------------------------
//      Modify the flag for next fault injection
//-------------------------------------------------------------

         if (bz<0)
          {
            pin=pin-1;
            bz=bz*(-1);
          }
         else
          {
           bz=bz*(-1);
          }
   }

 system(cpvec);//Recover the original vector file and end of input-fault-
injection.


//**********************************************************************
//               Function for wire-fault-injection
//**********************************************************************

   bz=-1;
   mi=0;
   for (wir=0;wir<wirenum;wir++)
    {
      if (bz<0)
          {
           strcpy(ijf,"0");
           for(i=0;i<30;i++)
            {
              if (wirec[mi]!=',')
               {
                w0[i]=wirec[mi];
                //i=i+1;
                mi=mi+1;
               }
              else
```

```
        {
         mi=mi+1;
         w0[i]='\0';
         break;
        }
      }
  }
  else
  {
   strcpy(ijf,"1");
  }
  printf("ijf is %s\n",ijf);
  printf("w0 is %s\n",w0);
   w1[0]=',';
   w1[1]='\0';
   w2[0]=',';
   w2[1]='\0';
   strcat(w1, w0);
   strcat(w1, ",");
   strcat(w2, w0);
   strcat(w2,")");
   printf("w1 is %s\n",w1);
   printf("w2 is %s\n",w2);

  tem1 = fopen(vlgbak, "r");
  tem2 = fopen(MUTname,"w");
  while ( fgets(aLine, MAX, tem1) != NULL )
  {
         if ( (ptr = strstr(aLine,"wire ")) != NULL )
           {
            wireline=1;
           }
         if ( (ptr = strstr(aLine,w1)) != NULL )
          {
            if(wireline==1)
             {
              wireline=wireline;
             }
            else
             {
              strcpy(line2, ptr + strlen(w1));
              *ptr = 0;
              strcat(aLine, ",");
              strcat(aLine, ijf);
              strcat(aLine,",");
              strcat(aLine, line2);
             }
           }
           if ( (ptr = strstr(aLine,w2)) != NULL )
            {
                if(wireline==1)
                {
                 wireline=wireline;
                }
                else
                {
                 strcpy(line2, ptr + strlen(w2));
                 *ptr = 0;
                 strcat(aLine, ",");
                 strcat(aLine, ijf);
                 strcat(aLine,")");
                 strcat(aLine, line2);
                }
```

```
                }
              if ( (ptr = strstr(aLine,";")) != NULL )
                {
                 wireline=0;
                }
                fprintf(tem2, "%s", aLine);
          }
        fclose(tem1);
        rewind(tem2);
        fclose(tem2);
        system(delspeed);
        system(cml);
        loopnum=loopnum+1;
        system(sim);


//*************************************************************************
//           Function for comparison of 2 result table files
//                       for wire-fault-injection
//*************************************************************************

        tem1 = fopen(tbl, "r");
        tem2 = fopen(tbak,"r");
        r=-1;
        s=-1;
        findfault=0;
        while ( fgets(aLine, MAX, tem1) != NULL )
         {
                if ( (ptr1 = strstr(aLine,"0>")) != NULL )
                {
                   r=r+1;
                   strcpy(aLine, ptr1 + 3);
                    while ( fgets(line2, MAX, tem2) != NULL )
                    {
                     if ( (ptr2 = strstr(line2,"0>")) != NULL )
                      {
                        s=s+1;
                        if (s==r)
                         {
                           strcpy(line2, ptr2 + 3);
                           if ((s>0)&&(strcmp(aLine,line2)!=0))
                            {
                              countfault[r]=countfault[r]+1;
                              strcpy(cpvt,"copy ");
                              strcat(cpvt,tbl);
                              strcat(cpvt," ");
                              strcat(cpvt,MUT);
                              strcat(cpvt,w0);
                              strcat(cpvt,ijf);
                              strcat(cpvt,"tbl.bak");
                              printf("cpvt is %s\n",cpvt);
                              system(cpvt);
                              findfaultnum=findfaultnum+1;
                              findfault=1;
                              break;
                            }
                            else
                            {  break; }
                         }
                      }
                     }
                   if (findfault==1)
                    {break;}
```

```
                }
            }

            fclose(tem1);
            fclose(tem2);

//---------------------------------------------------------------
//      Modify the flag for next fault injection
//---------------------------------------------------------------

          if (bz<0)
           {
             wir=wir-1;
             bz=bz*(-1);
           }
          else
           {
            bz=bz*(-1);
           }
    }

system(cpv);// Recover the original .v file and end of wire-fault-injection.


//***********************************************************************
//                Function for output-fault-injection
//***********************************************************************

    bz=-1;
    mi=0;
    for (pin=0;pin<outnum;pin++)
     {
       if (bz<0)
           {
            strcpy(ijf,"0");
           }
          else
           {
            strcpy(ijf,"1");
           }
           printf("ijf is %s\n",ijf);

          tem1 = fopen(tbak, "r");
          r=-1;
          while ( fgets(aLine, MAX, tem1) != NULL )
          {
                  if ( (ptr = strstr(aLine,".0>")) != NULL )
                    {
                      r=r+1;
                      if ((r>0)&&(r<m))
                      {
                        ptr=ptr+4;
                        ptr=ptr+(innum*2);
                        ptr=ptr+2;
                        ptr=ptr+(pin*2);
                        a[0]=*ptr;
                        printf("a0 is: %c\n",a[0]);
                        printf("ijf is:%c\n",ijf[0]);

                        if (a[0]!=ijf[0])
                          {
                             countfault[r]=countfault[r]+1;
                              findfaultnum=findfaultnum+1;
```

```
                              findfault=1;
                             break;
                           }
                        }
                     }

           }
            fclose(teml);
            loopnum=loopnum+1;

//-----------------------------------------------------------
//      Modify the flag for next fault injection
//-----------------------------------------------------------

           if (bz<0)
            {
              pin=pin-1;
              bz=bz*(-1);
            }
           else
            {
             bz=bz*(-1);
            }
     fclose(teml); // End of output-fault-injection.
    }

//***********************************************************
//           Function for generating a fault-report
//***********************************************************

   tblbak = fopen(tbak, "r");
   while ( (fgets(aLine, MAX, tblbak) != NULL)& (testno<(m+1)) )
     {
       if ( (strstr(aLine,toFind4)) != NULL )
        {
          ptr = strstr(aLine,toFind4);
          ptr=ptr+4;
             for (int i=0;i<2*(n+2);++i)
               {
                 if (*ptr != '=')
                  {
                   a[i]=*ptr;
                   ++ptr;
                  }
               }
            report = fopen(rpt, "a");
                 fprintf (report, "Test%d %s ",testno,a);
                 fprintf (report, " %d faults detected \n",countfault[testno]);
                 testno=testno+1;
            fclose(report);
          }
       }
     fclose(tblbak);

                 gettime(&t);  // Record the end time
                 t2hour=t.ti_hour;
                 t2minute=t.ti_min;
                 t2second=t.ti_sec;

           report = fopen(rpt, "a");
           fprintf (report, "findfaultnum is %d\n",findfaultnum);
           fprintf (report, "totalfaultnum is %d\n",(2*(innum+outnum+wirenum-2)));
```

```
            fprintf (report, "start time is %d",t1hour);
            fprintf (report, ":");
            fprintf (report, "%d",t1minute);
            fprintf (report, ":");
            fprintf (report, "%d\n",t1second);
            fprintf (report, "end time is %d",t2hour);
            fprintf (report, ":");
            fprintf (report, "%d",t2minute);
            fprintf (report, ":");
            fprintf (report, "%d\n",t2second);
            fclose(report);

 printf("loopnum is %d\n",loopnum);
 printf("wirenum is %d\n",wirenum);
 printf("innum is %d\n",innum);
 printf("outnum is %d\n",outnum);
 printf("findfaultnum is %d\n",findfaultnum);

 return 0;
}
// End of program.
```