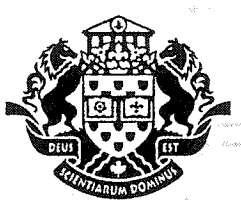

Université d'Ottawa · University of Ottawa

Université d'Ottawa • University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Liwu 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

Hardware and software co-desing in space compaction of cores-based digital circuit

S. R. Das

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

E. M. 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 POSTDOCTORAL STUDIES

Hardware and Software Co-Design in Space Compaction of Cores-Based Digital Circuit

BY

LIWU JIN

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE (SYSTEMS SCIENCE)

SYSTEMS SCIENCE

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

University of Ottawa

UNIVERSITY OF OTTAWA

MARCH 2004

Your file Votre référence
ISBN: 0-494-01505-5
Our file Notre référence
ISBN: 0-494-01505-5

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 for his support and encouragement.

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. Chuan Jin, who helped me immensely in successful completion of this work.

Finally, I owe very special thanks and gratitude to my wife Linyan Deng, for her love and never-ending support.

ABSTRACT

Implementation of fault testing environment for embeded cores-based digital circuits is a challenging endeavor. The subject thesis aims developing techniques in design verification and test architecture utilizing well-known concepts of hardware and software co-design. There are available methods to ensure correct functionality, in both hardware and software, for embeded cores-based systems but one of the most used and acceptable approaches to realize this is through the use of design for testability. Specifically, applications of built-in self-test (BIST) methodology in testing embeded cores are considered in the thesis, with specific implementations being targeted towards ISCAS 85 combinational benchmark circuits. Experimental results provided in the thesis prove the validity and importance of the approaches proposed for the design verification and test based on hardware and software co-design concepts utilizing Altera MAX Plus II simulation environment.

TABLE OF CONTENTS

List of Acronyms

ASIC Application Specific Integrated Circuit

ATE Automatic Test Equipment

ATPG Automatic Test Pattern Generation (or Generator)

BIST Built-in Self-Test

CAD Computer-Aided (or Assisted) Design

CMOS Complementary MOS

CPU Central Processing Unit

MUT Module Under Test

DFT Design for Testability, or Design for Test

FPGA Field Programmable Gate Array

FT Functional Test (or Tester)

HDL Hardware Description Language

IC Integrated Circuit

I_{DDQ} Quiescent Current in CMOS

IEEE Institute of Electrical and Electronics Engineers, Inc.

I/O Input / Output

K 10^3

LAN Local Area Network

LFSR Linear Feedback Shift Register

LSI Large Scale Integration

MOS Metal-Oxide-Semiconductor

MUX Multiplexer

NS Nanosecond

PCB Printed Circuit Board

RAM Random-Access Memory

RCU Response Compaction Unit

ROM Read-Only Memory

RS Set/Reset

s-a-0 Stuck-at-0

s-a-1 Stuck-at-1

SPICE Simulation Program with Integrated Circuit Emphasis

SSI Small Scale Integration

TPG Test Pattern Generation (or Generation)

TTL Transistor-Transistor Logic

ULSI Ultra Large Scale Integration

VHDL Very High Speed Hardware Description Language

VLSI Very Large Scale Integration

Z High Impedance

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Since William Shockley, John Bardeen, and Walter H. Brattain of Bell Laboratories invented the first primitive transistor in 1948, digital circuit technology has gone a long way in terms developmental progress. In particular, the growth of integrated circuit technology (IC) has progressed exponentially in integration scale. The IC technology has gone through the milestones of small-scale integration (SSI), medium-scale integration (MSI), large-scale integration (LSI), and very large-scale integration (VLSI).

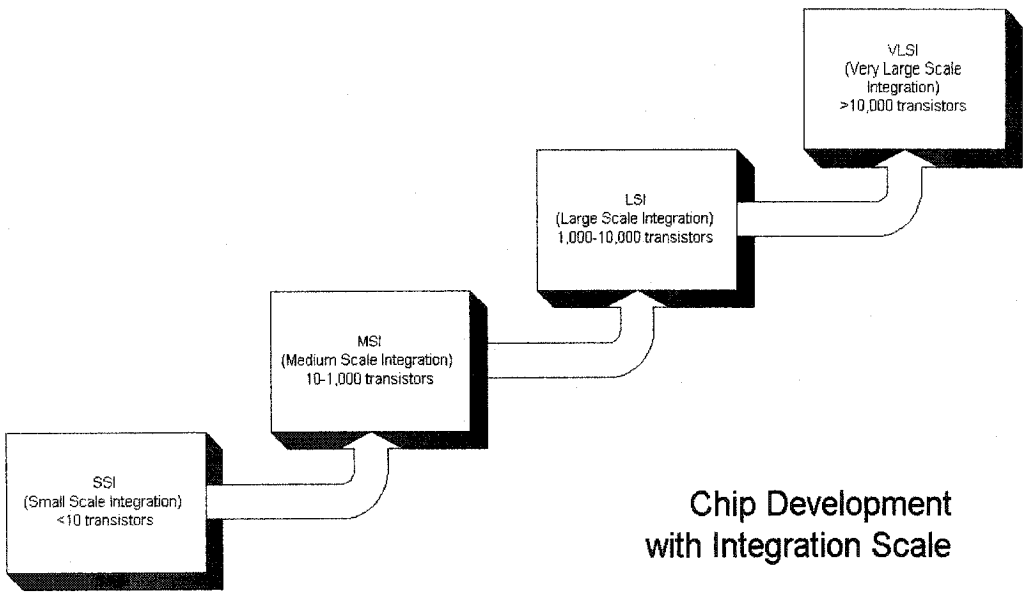

Figure 1 Chip development history

Figure 1 shows the integration scale in one single chip. The maximum number of transistors on a chip approximately doubles every eighteen months. The rule is commonly known as Moore's law, which was predicted by Gordon Moore in 1965, and is shown in Figure 2.

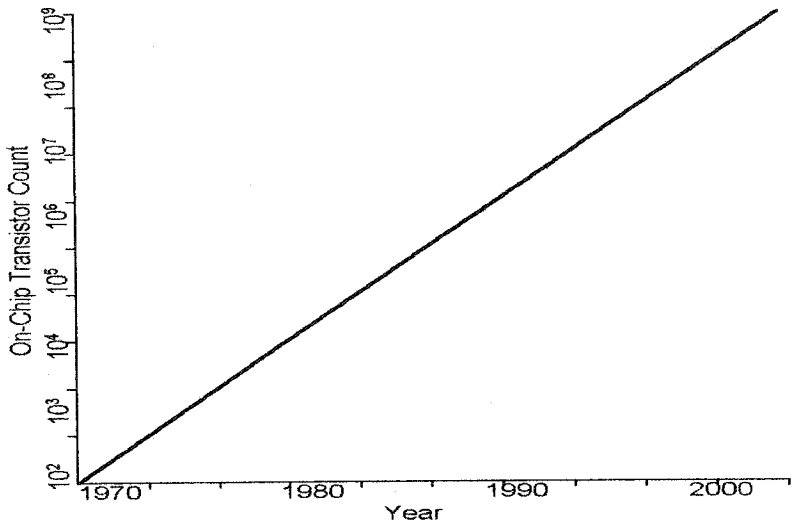

Figure 2 Moore's law

The number of devices that can be integrated into a chip has been increasing explosively with advances in VLSI technology. This has enabled the development of very complicated systems design using a relative small number of chips and boards. With the development of modern SOC (System-on-a-Chip) technology, it is possible to integrate a series of modules (cores) into an individual chip. Meanwhile, it has led to large amounts of time and expense being devoted to ensuring that complex systems are free of defects or failures.

The IC technology involves in many aspects such as logic design, modeling, simulation, testing, and fabrication. With increasing density of integration in digital circuit design, digital circuit testing has assumed significant importance. Verifying that the system behaviour is correct has become a very challenging task, requiring more effective and appropriate testing methods to guarantee correct performance of digital systems.

The type of circuits to be tested can be classified as follows: analog, digital, or mixed-signal circuits. Analog circuits have the characteristics that the domain of values of the input and output signals is analog. Digital circuits have the property that the scope of values of the input and output signals is digital (binary). Mixed-signal circuits are combined with analog and digital signals in the input and output. Testing mixed-signal

circuits is based on a combination of analog and digital test techniques. This thesis focuses on testing digital circuits.

In this Chapter, we first introduce some basic concepts in testing. A test is a procedure to distinguish between good and faulty parts. The term fault refers to physical difference between a correctly operating circuit with its faulty counterpart. A fault may result in an error in the system. The error could cause a whole system failure eventually. Note that a fault does not certainly (or immediately) result in a failure in a system. Here, a system means chip, array, or board.

Faults can be categorized as permanent faults and non-permanent faults. Permanent faults exist in a system permanently due to incorrect connection, short circuit, and other physical defects. Non-permanent faults are present only for a shot time. Since these faults occur at random (transient or intermittent faults, for example), these may or may not affect the function of a system. Obviously, the detection for permanent faults is easier than those of the non-permanent faults.

The aim of testing is to distinguish between good and faulty parts. The fault diagnosis problem is classified as being comprised of fault detection and fault location. Fault location is more difficult to accomplish based on the complexity of circuits to be tested.

Concurrent tests refer to testing that can be executed during normal operation of circuit. It is mainly used in fault-tolerant computers for self-testing. Non-concurrent tests are tests that cannot be performed during normal operation. The advantage of this method is that it is able to detect and/or locate more complicated faults. Non-concurrent fault detection is the subject of this thesis.

Design for testability (DFT) is a technique that facilitates non-concurrent tests to be executed faster with higher fault coverage by adding extra circuitry on the system to-be-tested. Up to now, many methodologies of design for testability have been developed to enhance testing ability.

A widely accepted approach is BIST (built-in self-test) to cope with the testing issue at the chip level. There are two modes for BIST test. One is on-line BIST in which the chip

tests itself under normal operation. Off-line BIST tests the chip under a test mode. A simple BIST structure is shown below.

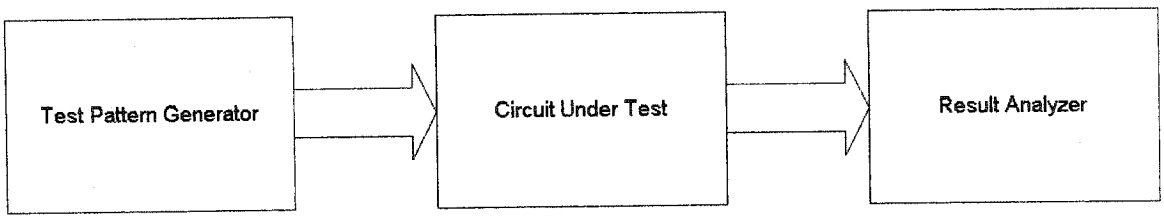

Figure 3 A simplest BIST structure

Test pattern generation is the important part of BIST approach. The main function of the test pattern generation is to apply specific test patterns to MUT (module under test). There are several approaches to generate test patterns for MUT. The test patterns could be exhaustive, pseudoexhautive, pseudorandom, and deterministic compact test sets.

In conventional testing methods, test patterns are normally generated externally by applying CAD (computer-aided design) tools. Some expensive ATE (automatic test equipment) is used to exercise the MUT and to determine whether the responses match those of the fault-free responses of the module, which are stored in memory. With the high level of integration in a chip these days, testing expenses thus increase rapidly, requiring need of more complex ATE (the cost of such a testing equipment could well exceed 2 million US dollars), thereby resulting in much more testing time. To overcome these disadvantages, some approaches were developed to increase the controllability and observability of chip testing, making test pattern generation and fault detection easier. [42]

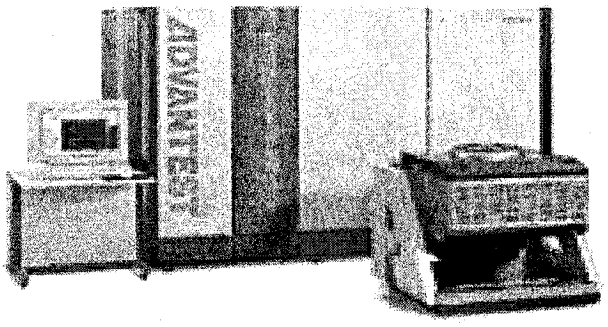

Figure 4 Automatic test equipment

In exhaustive testing approach, all possible input test patterns will be applied to the module under test. Suppose we have a combinational logic circuit to test. If this circuit has N inputs, there are 2^N possible input combinations. A simple testing approach is to apply each of these input test pattern combinations in turn. Then we check the resulting output. For large circuits, exhaustive testing approach is inappropriate since it takes too much time and resource. Actually, exhaustive testing approach for logic circuits is seldom to be invoked.

The main advantage of this approach is that all possible faults can be detected except bridging faults. Obviously, it is appropriate for small circuit modules since the test patterns to be dealt with are not that many. As the number of inputs increases, the testing time becomes intolerable. The specific testing equipment will take years to run to test a circuit module with 64 inputs. Therefore, exhaustive testing is feasible for circuits with only small number of inputs (normally no more than 20~25).

An improved approach is pseudoexhaustive testing. The methodology is to partition the module under test into several sub-modules with proper inputs. Each sub-module is applied with exhaustive test patterns. This approach completely retains the advantage of exhaustive testing and significantly decreases the testing time.

Pseudorandom means that it is adequately random close to being truly random. It is important to note that no test is completely random since there is no ideal random number generator. However, it is fairly easy to generate pseudorandom test sets.

Pseudorandom testing is widely employed in practice since it generates less test patterns that apparently reduce the testing time and cost.

To understand implementation of pseudorandom testing, some concepts shall be emphasized here. One is that of fault coverage that refers to the percentage of faults that can be detected. The next is test pattern number that means how many test patterns will be applied in a module under test. If the number is too small, the fault coverage will be very low. On the other hand, if the number is too high, the testing time will be prohibitive similar to exhaustive testing.

Pseudorandom testing is extensively implemented for large circuits. A pseudorandom test sequence is a sequence of numbers generated mathematically. For the deterministic test patterns, these can be stored on chip as fault-free responses. These can be used for comparison with the responses of the MUT. However, the storage of a large set of test patterns increases the chip area and overhead. A compression module significantly reduces the chip area and reduces overhead. The compaction module can be used to compress all responses into a single vector (signature) so that the comparison can be done easily. The compression can be divided into space compression and time compression. We will discuss it later in details.

We normally use LFSR (linear feedback shift registers) to generate pseudorandom test patterns and also as response compressor. Figure 5 shows the general structure of a linear feedback shift register.

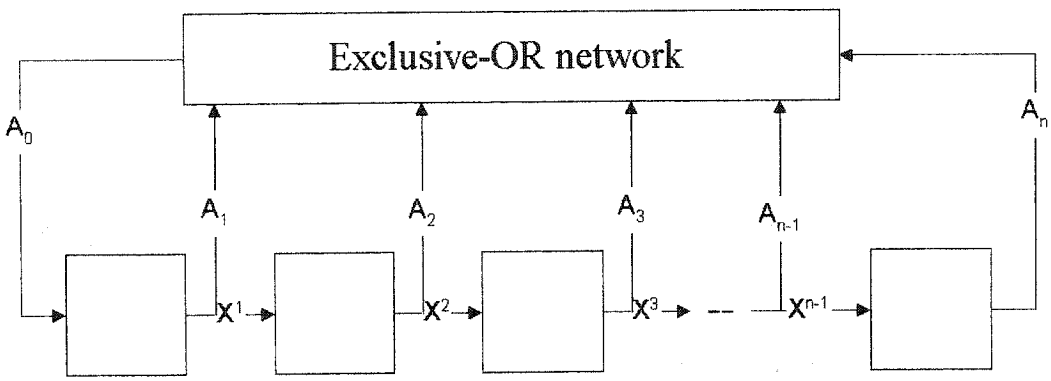

Figure 5 Linear feedback shift register

Each block represents a D flip-flop. A common clock synchronizes all flip-flops. The Exclusive-OR network in the feedback path performs modulo-2 addition of the values (x's) in the flip-flops. A characteristic polynomial represents a linear feedback shift register with n stages as follows.

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_2 x^2 + a_1 x^1 + a_0$$

For example, to implement $P(x) = x^3 + x + 1$, the settings of this linear feedback shift register should be $a_3=1$, $a_2=0$, $a_1=1$ and $a_0=1$. This linear feedback shift register will produce a sequence of 7 non-zero binary patterns such as: 001 -> 100 -> 110 -> 111 -> 011 -> 101 -> 010. [8]

As mentioned earlier, BIST technique usually combine compression module with the MUT to reduce the storage of corresponding response data. Several compression techniques have been proposed in recent times. These include transition count, syndrome checking, signature analysis techniques, etc.

The transition count technique counts the total number of transitions from 1 -> 0 and 0 -> 1 in a response sequence. For example, if a response sequence S = 100110110, then the transition count c (S) = 5. Therefore, we only record the transition count number for a given input sequence instead of storing the entire output response sequence. This number is then compared with the expected one. [16] If they match, the MUT is fault-free; otherwise, it is faulty.

The length of the signature must be as short as possible in order to minimize the amount of memory required to store the fault-free responses. A fundamental problem with compression techniques is error masking or aliasing [1-4], which occurs when the signatures from faulty output responses map into fault-free signatures. Aliasing results in loss of information, which in turn affects the test quality of BIST and reduces the fault coverage.

The main advantage of transition count technique is that it significantly saves space to store entire response sequence. However, this technique results in many fault-masking errors. For example, the transition count c (S) = 1 can be generated by the following response sequences: S1 (0111), S2 (1000), S3 (0011) and S4 (1110). It causes many faults undetected, and this results in what is called error masking or aliasing. A good compression technique should minimize this masking effect as much as possible.

The term syndrome is defined as $S = K/2^n$, where K is the number of minterms realized by the circuit module (normally we take the minterm as 1) and n is the number of input lines [9]. Because the syndrome is a functional testing, various realizations of the same function might have the same syndrome. Faults are detected by comparing the output syndrome with the stored fault-free syndrome.

The signature analysis technique uses a unique data compression technique to reduce long response sequence into a unique code called the signature. Signatures can be generated by feeding the sequence into an n-bit LFSR (linear feedback shift register). Signature analysis is widely used in digital circuits testing. A signature analyzer is digital equipment to detect errors in data streams caused by the faults of module under test. [8]

A conventional signature analysis scheme consists of a shift register and a modulo-2 adder. The block diagram is illustrated in Fig. 6.

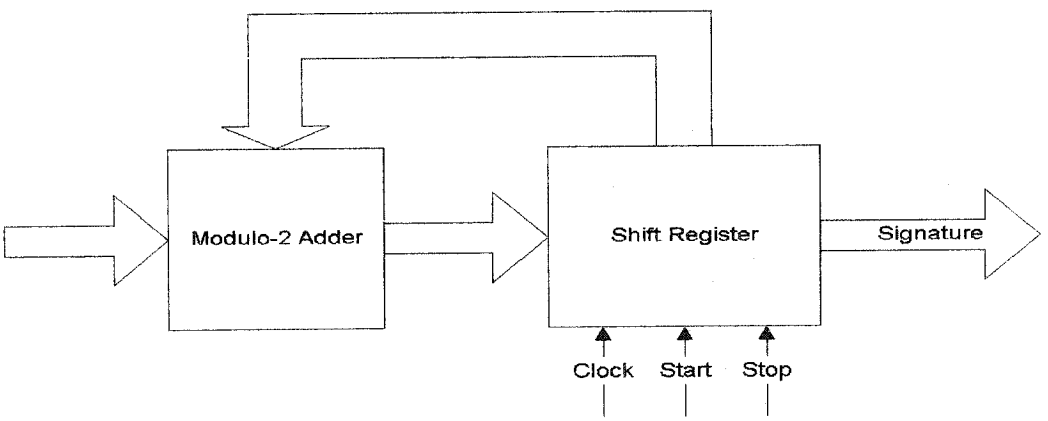

Figure 6 Signature analyzer block diagram

A signature signal is a polynomial generated by a shift register. The signature analyzer control signals are "clock", "start" and "stop". "Clock" signal provides a standard frequency to signature analyzer. "Start" and "stop" signals set a time window within which data compression is performed by the signature analyzer.

The complete mathematical structure of signature analyzer can be presented as

$a_i(0) = 0$ i=1 to m

$a_1(k) = y(k) \oplus \sum_{i=1}^{m} \alpha_i a_i(k-1)$

$a_j(k) = a_{j-1}(k-1)$ j=2 to m, k=1 to l

where l is normally taken to be equal to or less than 2m-1; hence it determines the length of sequence to be compressed. The binary code can be mapped into hexadecimal code and fixed on its memory elements.

Currently signature analysis is widely used in time compression in testing digital systems. The principle of signature analysis is that long output responses of digital circuits are represented by short keywords called signatures.

A high-level verification environment consists of a mixed-level hardware and software co-simulator, test pattern generators, compressor, and comparator. The verification of a specific digital design is actually a process of ensuring that the logical design of the circuit module satisfies the required architectural specification.

A theoretical verification environment to be used is shown as follows. In practical implementation, some modules might have to be modified to fit the simulation environment. The details of the procedure to be used will be stated later.

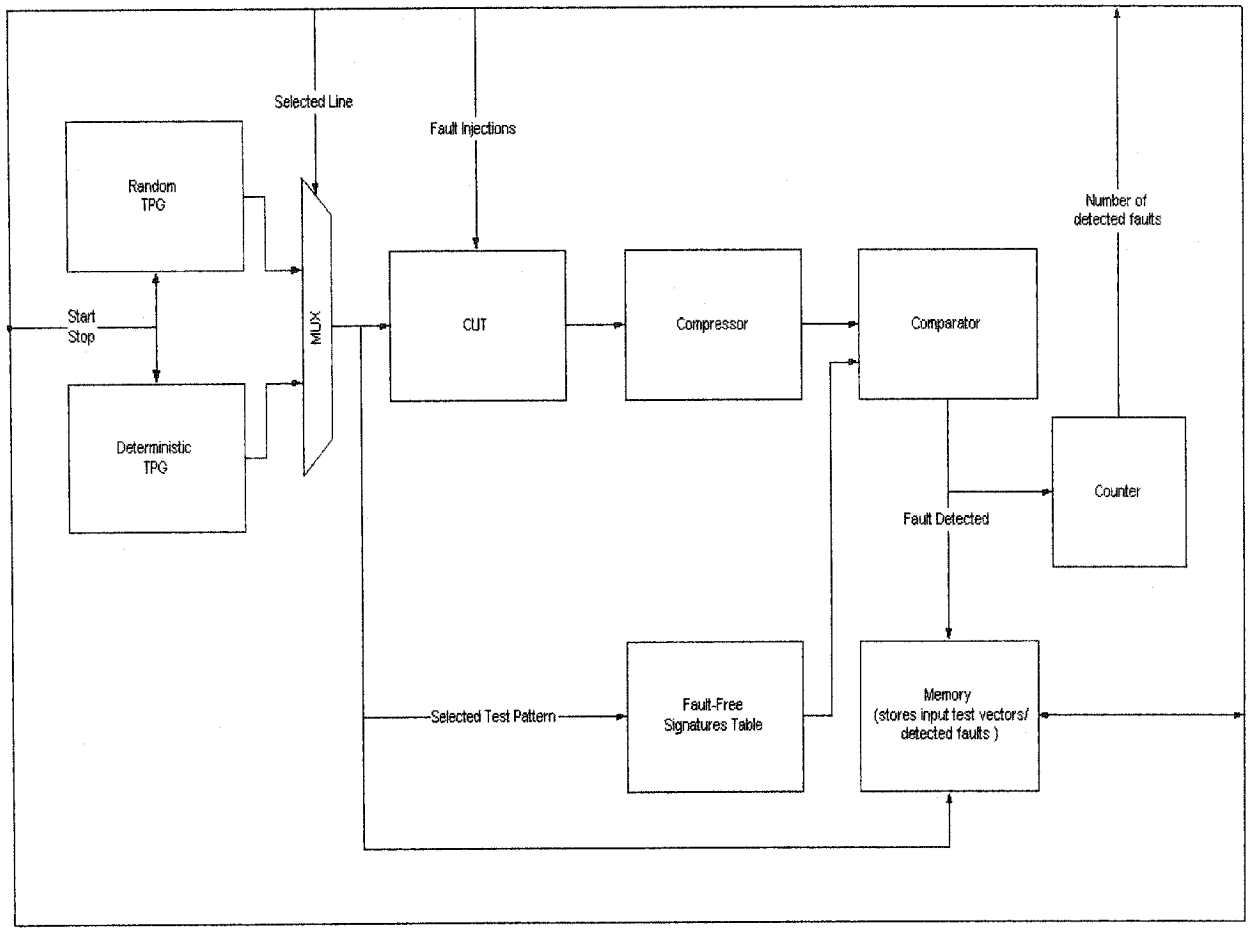

Figure 7 The hardware verification scheme

Before implementing a system or a prototype (circuit model), simulation is a very important process to emulate the complete process of the system. However, building and testing a system model is also time- consuming and expensive. Extensive efforts were devoted to optimize the testing process, viz. Incorporation of space and time compression techniques, which can be used to reduce testing time and memory space, thereby, obviously reducing the testing time.

The thesis is divided into several Chapters. Chapter 1 emphasizes the need and history of digital circuit verification and points out the limitations of conventional approaches to diagnosing failures of digital systems that employ large-scale and very large-scale integration. We briefly describe the theory of testable design for digital circuits. The concepts of testability and testability criteria have been introduced, and

approaches of hardware and software co-design and that of response data compression have been discussed.

The next chapter discusses some important concepts in the context of our system implementation. For example, the BIST structure we emphasize here will be employed for our main testing framework subsequently. The response compaction methods, which are extensively used for testing complex digital circuits, are described in detail.

Chapter 3 presents the implementation of digital circuit verification in detail. It also covers the methodologies and algorithms. The design procedures for a pseudorandom test sequence generator are discussed for both random and pseudoexhaustive testing.

The experimental results are analyzed in Chapter 4 to validate the correctness of the proposed approaches. Analysis of test results is done to evaluate the efficiency of testing procedure.

The last chapter provides conclusions and future work we will concentrate on, and on some new methodologies to be developed later.

CHAPTER 2

Digital Design Verification Scheme

1. FAULT MODELS

Circuits can be described at different levels of abstraction in the design hierarchy as follows: behavioral, functional, structural, switch-level, and by geometric description. This process of fault modeling considerably alleviates the test generation complexity.

Low (high) voltages in digital circuits can be represented by logic value 0 (1). Accordingly, logic value "x" refers to unknown status and logic value "z" means high-impedance state. Combinational circuits with defined input patterns have unique logic values at each internal node and definite output responses.

When a combinational circuit is powered-up, the output responses of the circuit can be determined after a very short period of time. The value "x" appears at the initial state of sequential circuit since the initial state of memory elements such as flip-flops is usually unpredictable. Nets are a kind of physical connections without storing any value. If not driven by a driver (i.e., gate), their value is "z".

Faults in a digital circuit result from defective components, shorts to voltage source or ground, signal delays, etc. Faults refer to the effect of the physical faults on the operation of the system. Therefore, they can be represented by a model, which allows a mathematical methodology of testing and diagnosis. Three major fault models are commonly used. There are stuck-at-fault, bridge fault and stuck-open fault models.

Fault models are used to analyze the result of testing. The most popular model is the stuck-at-fault model. It can be divided into single stuck-at-fault and multiple stuck-at-fault models. The single stuck-at-fault model is the most common model used to represent logic faults since the experience shows that 70% of the possible circuit faults can be accounted for by using a single stuck-fault model [10]. Our major effort will focus on fault detection and diagnosis of digital circuits based on such single stuck-fault model.

The single stuck-fault model assumes that only one fault can exist at any given time in a logic circuit. A stuck-fault results from a wire being accidentally connected to either ground or a voltage source. When attached to ground, a wire's value will be stuck-at logic 0 (stuck-at-0).

Similarly, when a wire is connected to a high voltage source, that wire's value will always be stuck-at logic 1 (stuck-at-1). Stuck-at-1 and stuck-at-0 faults are often abbreviated as s-a-1 and s-a-0, respectively.

The stuck-fault model is also used to represent multiple faults in circuits. It assumes that more than one stuck-at-fault exist in a logic circuit at the same time. For a circuit with n wires, the possible number of multiple faults is 3^n-1, and as such, for large values of n, it is almost impossible to find test sets for multiple faults in a given circuit.

Bridging faults can occur when wires become physically connected. The resulting values on the wires depend on the original signal values.

Fault modeling was first developed at logic gate level. Most of the solutions and techniques that were developed are specifically targeted to circuits at the gate level. Our fault modeling for verification and implementation is also based on circuits at the gate level.

2. VERIFICATION SCHEME

The verification scheme involves test pattern generation, faults modeling, corresponding outputs storage, and comparison with predefined fault-free signatures to verify whether the faults are detected or not.

The objective in a test scheme development for a digital circuit is to determine the smallest set of test vectors necessary to verify the correctness of the circuit functions. The test vectors could be deterministic exhaustive, pseudoexhaustive, or compact, or even pseudorandom. BIST techniques normally utilize pseudorandom or pseudoexhaustive test patterns, or on-chip storing of reduced test sets.

In a large circuit or chip, stuck-faults can occur in any wire. Actually, it is quite difficult to detect these faults. But quality control systems provide verification scheme to more effectively to detect these faults in a digital circuit. Figure 8 shows a specific MUT (Module Under Test) in details.

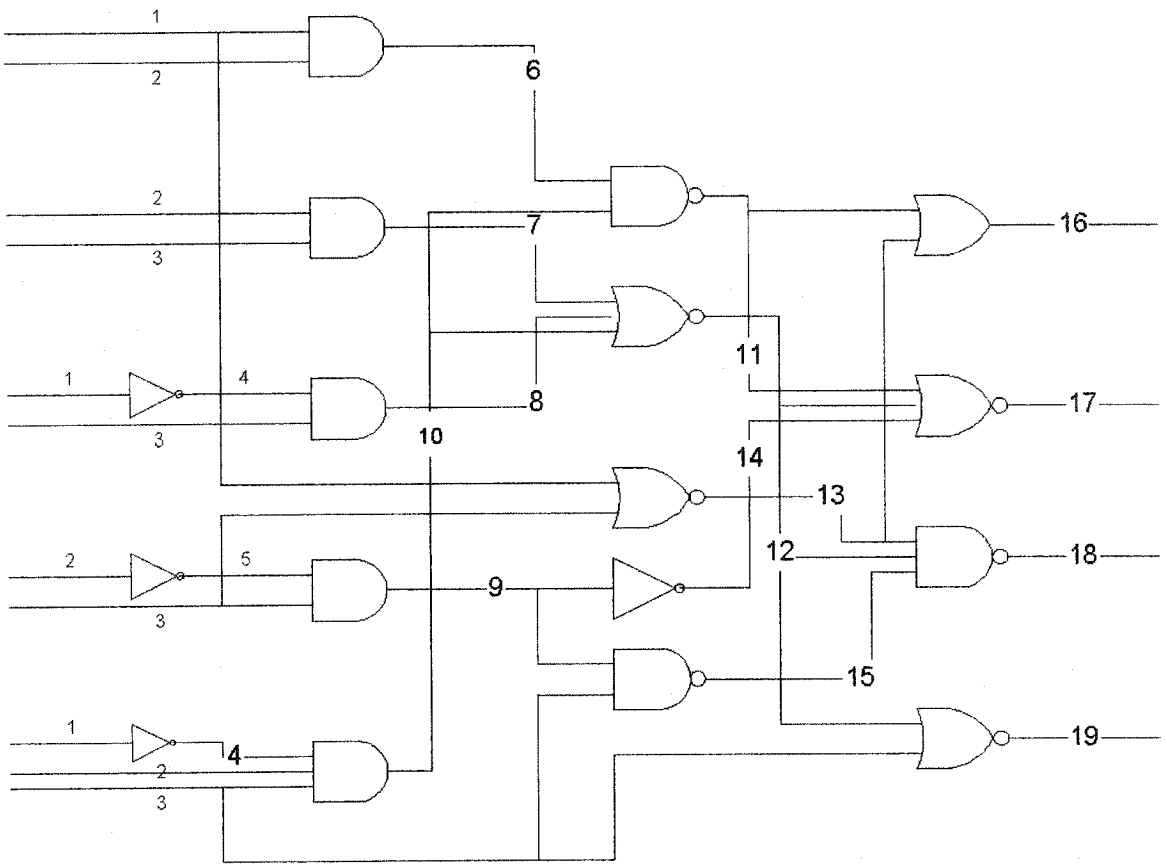

Figure 8. A simple circuit module under test

There are seventeen gates in this preliminary circuit module: four inventers, five AND gates, three NAND gates, four NOR gates, and one OR gate. Moreover, we have thirty-four separate wires, which interconnect these gates. But in real practice, we only consider nineteen mutually exclusive wires and eliminate the others. Accordingly, we have got three inputs that are labeled as 1, 2, 3 and four outputs that numbered as 16, 17, 18 and 19.

The main point to emphasize again is that these wires must be mutually exclusive though sometimes these are split into two wires. Since the fault injection will be based on

mutually exclusive wires, only one specific fault can be injected on that wire at any one time.

Our particular MUT has been designed for testing. Any injected stuck-at-fault can be detected even if it is in an internal wire.

3. OUTPUT COMPRESSOR

The output compression is utilized to reduce the number of system outputs without compromising the MUT's fault detection capabilities. Especially, in VLSI circuits or chips, extensive testing requires more memory to store test patterns and results. Output compressor will obviously decrease storage needs for memory and speed up the testing procedure. In Fig. 9, there are three different compressor circuits. They are all aliasing-free which means all faults can be detected at the outputs of the MUT and the compressor combined. In practice, the output compressors reduce the number of system outputs in the present case from four to two.

The output compression approach utilized to design the compression networks makes use of switching theory concepts such as Hamming distance, sequence weights, cover table and frequency ordering, together with those of Nth-order missed error probability estimates under stochastic dependence of line errors in the design based on detectable single stuck- line faults of the MUT. Here, we will focus on aliasing-free space compaction. The advantages of this approach are that it achieves zero-aliasing without any modifications of the MUT and the area overhead and signal propagation delay are less compared to linear parity tree space compactors. Moreover, the approach uses less computation in the process of design.

The output compressor can be used in the final hardware circuit realization of the verification scheme. As shown in Fig. 9, Type 1 is the simplest with least delay time. Type 2 and Type 3 compressors will be implemented after successful implementation and complete testing of the first one.

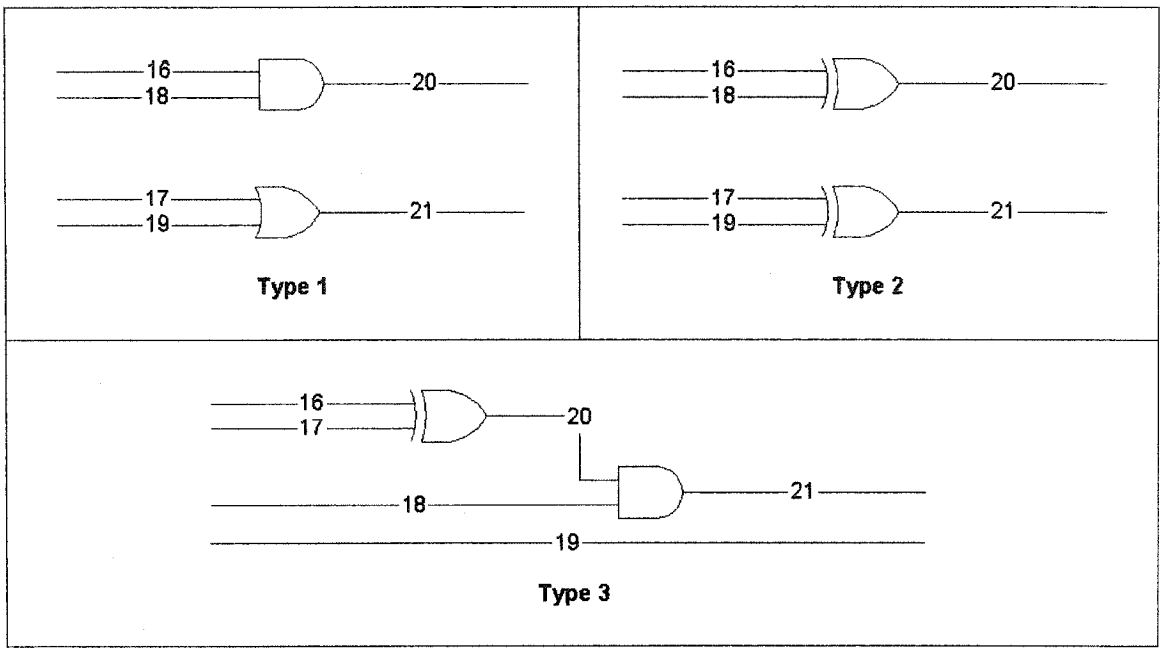

Figure 9. The preliminary compressor circuitries

The response compaction unit in BIST can be divided into a space compression unit and a time compression unit. Generally, the output responses coming out of an MUT are first fed into a space compressor followed by a time compactor. The length of output bit stream coming out of the time compressor is much shorter than its original length. In space compression, test responses are usually compressed into only one sequence. [53]

Space compression refers to the process of reducing a r-bit data stream to a q-bit data stream, where $q < r$. Space compression provides an effective way to achieve high quality built-in self-testing of complex circuits with less testing time and area overhead. It merges test sequences coming from a module under test preferably into a single bit stream (length r). This single bit stream is then fed into a time compression unit. Eventually, a compressed sequence length q ($q < r$) is formed at the output.

The fundamental issue with compression techniques is error masking or aliasing, which occurs when the signatures from faulty output responses map into fault-free signatures, eventually affecting the fault detection capability of BIST. [55]

4. Logic Fault Simulation for Combinational Circuit

Before implementing fault simulation, it is necessary to provide assumptions to be used to make analysis feasible. Since we choose single stuck-fault model, we assume that there is only one fault existing at one time in a circuit. It simplifies fault modeling and facilitates fault analysis.

Fault simulation is an important aspect in the digital circuit design process and is employed to detect a set of faults in a fault-injected circuit that are covered by a set of test vectors. The aim of fault simulation is to simulate a specific circuit with a given set of faults and to identify the test vectors that would detect those faults. The most important task is to decide whether a fault is detected by a particular set of test patterns.

For each test pattern, the good circuit is simulated at first or the right result is given. Then each circuit node is considered in sequence and injected with stuck-at-0 or -1 (s-a-0 or s-a-1), and the faulty circuit is then simulated.

Single fault propagation takes advantage of the small difference between the fault-free and faulty circuit. After the fault-free circuit is simulated, the target fault will be directly injected at the fault site. Signal changes are propagated from the fault site towards the circuit outputs. If the output responses of a faulty circuit differ from those of the good one, it means the corresponding fault is detected, and the fault is marked as detected. The process is repeated for all the nodes of the circuit being set to 0 or 1 until all the nodes have been tested. All detected faults are then divided by the total number of nodes in the circuit to give the percentage fault coverage. The type of fault analysis is called sequential fault grading method [1].

In practical all situations, the gate level simulation is widely adopted. Since we focus on logic simulation, the timing behavior of the circuit will not be considered. Therefore, timing models and specification of delays associated with circuit components are not taken into account.

The more the number of applied test patterns are, the higher will be the fault coverage achieved. At the end of the simulation run, there normally remain few hard-to-detect faults that only stimulate a low circuit activity.

Hardware fault injection is the core of fault simulation. In order to insert a selected fault on a specific wire without changing the structure of circuit, a method is introduced as shown in Fig. 10.

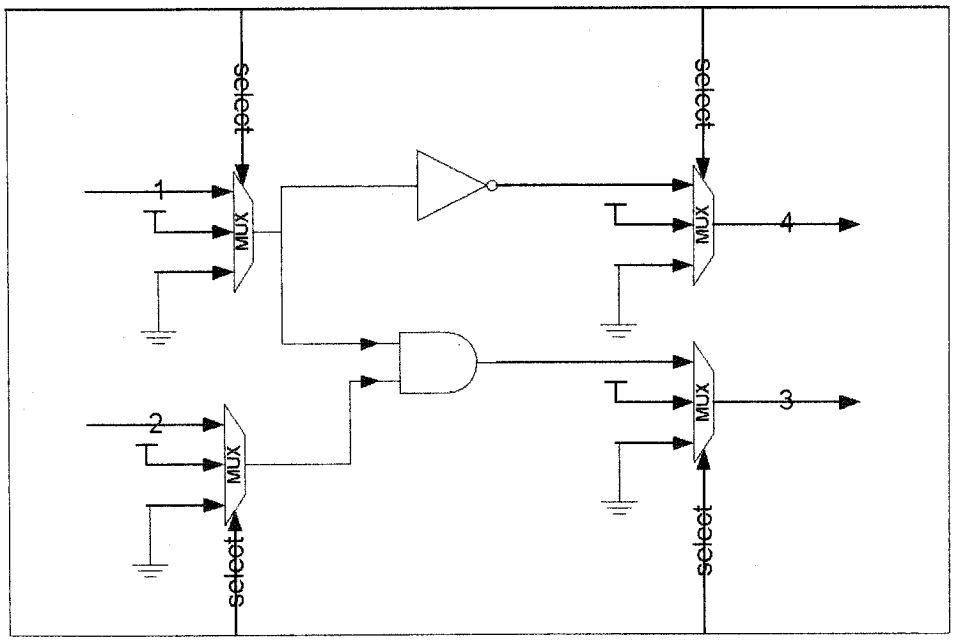

Figure 10. The hardware fault injection scheme

According to the figure, every mutually exclusive wire is interposed with a multiplexer. This multiplexer has three input lines which connect normal signal, stuck-at-1, and stuck-at-0 faults signal conditions. Which line will be connected depends on the 'select' signal. If the value of the 'select' signal of any multiplexer is at "00", then the wire will carry signal as it was. On the other hand, if the value is at "01", a logical 1 signal goes through the multiplexer. It means we inject a stuck-at-1 fault to this wire. Accordingly, if the value is at "10", a stuck-at-0 fault has been inserted to this wire through the multiplexer. Finally, if the value is "11", then we will again assume normal operation of this wire.

We plan to design a general multiplexer that will allow us to inject the faults at any wire. It means that we can use this multiplexer to inject stuck-at-faults with change of the structure of circuit. Table 1 illustrates the truth table associated with any multiplexer of the type shown in Figure. 10.

SelA	SelB	WireIn	WireOut
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Table 1. Truth Table for Fault-Injecting Multiplexer

We invoke simple K-map expansion and processing, and devise the gate level circuit schematic for any of the fault injection multiplexer as shown in Fig. 11.

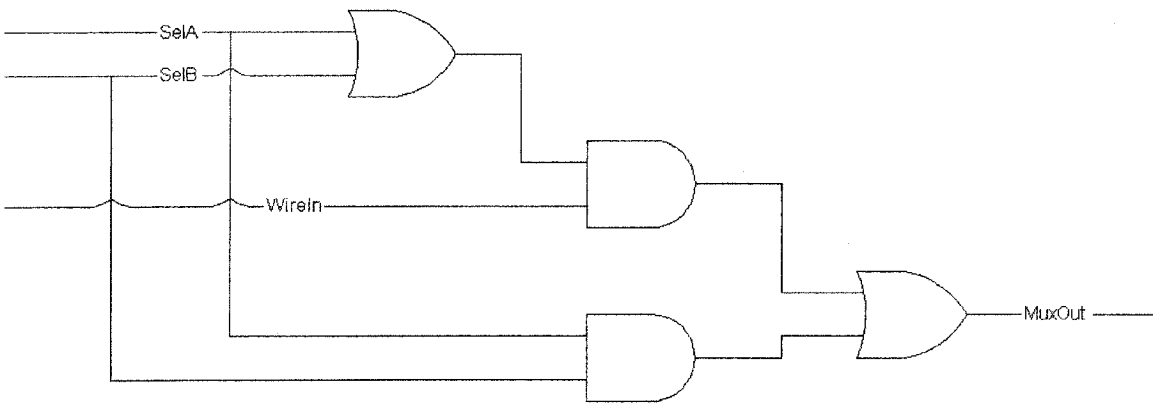

Figure 11. The fault injecting multiplexer

In order to insert a specific fault at a particular wire, we have to divide this wire into two parts. The forepart of this wire connects to 'WireIn' line of multiplexer. Meanwhile,

the 'MuxOut' line will link to the real part of the wire. The select signals will control which signal can be generated at 'MuxOut'.

5. BIST AND DFT

The purpose of digital circuit testing is to detect any failure or potential errors that will induce system malfunction. Actually, there are two major causes of failure, design errors and fabrication defects. In order to detect these faults in digital circuits, IC design methodology must include an important concept of design for testability (DFT).

Design for testability (DFT) is an essential part of the design process of a complex circuit. In a word, DFT involves embedding into the circuit some additional feature to enhance the controllability and observability of the circuit.

Controllability and observability are two key aspects to design for testability. Controllability refers to the ability to alter the state of every internal node in circuit. It requires its internal nodes to be accessible so that test signals can be inserted. Observability is the ability to monitor the state of any node in circuit, so that we can access correct testing data we need.

Design for testability has to be taken into account throughout the design process for a complete digital circuit product. To accomplish this objective, many approaches have been developed to improve the testability of digital circuits. For example, boundary scan models are embeded into an integrated circuit to facilitate its test, maintenance, and support. Boundary scan approach has been standardized in IEEE 1149.1, which specifies a standard interface to provide testability.

There are three approaches that are ad-hoc testing, scan-based approaches, and built-in self-test (BIST) testing. Here, we will focus on BIST technique.

One aspect of design verification is obviously to detect the faults in the circuit. The verification scheme not only includes generating faults for simulation, but also recording

the corresponding circuit outputs. Comparing the circuit outputs with predefined fault-free values, we can identify whether or not faults have occurred.

In a typical test environment, we need a test pattern generator (TPG) to generate the stimulus to apply to the module under test (MUT) for the detection of faults. However, with increasing of complexity of digital circuits, this approach is not suitable. Built-in self-testing (BIST) provides solution that can overcome many of the shortcomings of standard testing techniques. Built-in self-testing (BIST) is a design approach that combines concepts of both built-in test (BIT) and self-test (ST).

BIST refers to designing circuits with additional logic that can be employed to test proper operation of the primary (functional) logic. BIST can be on-line or off-line, concurrent or non-concurrent, and can be used in functional and structural mode. On-line testing can be done during normal functional operating conditions of the MUT. On the contrary, off-line testing is run when a system is not carrying out its normal functions. Obviously, real-time error detection is not important to off-line testing mode.

BIST techniques utilize pseudorandom, pseudoexhaustive, or even exhaustive test patterns or sometimes on-chip storing of reduced test sets to cover as many faults as possible in the MUT. The basic principle is to use a TPG that sends stimulus to the MUT. A test data analyzer captures these output streams. The faults are detected by comparing with fault-free outputs.

The test data analyzer consists of a response compaction unit (RCU), storage for the fault-free response of the MUT and a comparator. The RCU is comprised of two parts: space compression unit and time compression unit [1].

Since bit-to-bit comparison is time-consuming, in practice, output bit streams are compressed into "signatures". Compression techniques involve transition count, parity checking, and syndrome checking as well as signature analysis. An essential problem of compression techniques is error masking or aliasing [2], which introduces information loss and reduces the fault coverage.

CHAPTER 3

REALIZATIONS OF HARDWARE AND SOFTWARE CO-DESIGN

1. INTRODUCTION

A comprehensive testing system is synthesis of hardware, software, and test methodology that have evolved to form a solid basis for the test environment. The testing system must be extended to embed built-in test technology into systems for high testability.

The test hardware can be a collection of test circuits, power supplies, measuring devices, and transition devices that have digital-to-analog and analog-to-digital functions. Buffer memory will be used to store test patterns and output responses. Moreover, the compression network as an additional testing circuit exists as part of test hardware. Generally, these can be described by hardware description language, such as VHDL, Verilog HDL, or other types of hardware description languages. In this research, Verilog HDL is used to describe the hardware.

Test software is to realize the automation of computerized testing process. The programs embeded in a chip can control the functions of testing, applying test patterns to its specific ports. [57] Moreover, it also compares the output responses with stored fault-free responses. The comparison results are recorded for further analysis. Briefly speaking, test software will complete test automation and test simulation.

The work in the thesis aims digital design verification through implementation of relevant hardware and software. The design process is based on embeded cores-based system-on-chip [5] and is based on well-known principles of hardware and software co-design. The objective primarily is to implement a combinational logic circuit test simulation environment [3], and to realize the verification scheme, and to compare the results to theoretical ones, in order to prove that the test mechanism is working.

There exist many methods to make sure that the design is functioning correctly in hardware and software, but one of the most efficient and reliable ways is the design for testability approach. Our objective is to ensure that all detectable faults at the outputs of

the MUT are captured at the outputs of the test models (MUT and compressor). The current research attempts to extend these basic principles to accomplish a design that works in different operational environments.

The program is designed to handle ISCAS 85 combinational benchmark circuits, which is described by gate-level Verilog HDL. Hence, it is limited to only handle combinational or acyclic digital circuits with no feedback (or feedforward) loops in the structure. The testing environment is also based on ISCAS 85 combinational benchmark circuits. It uses both hardware pseudorandom input test pattern generator and saved deterministic test vectors.

Developmental environment includes Altera MAX PLUS II with Verilog HDL and C/C++ programming language. A software hardware co-design methodology is to be applied to synchronize the on-board processor and configurable logic blocks.

We choose Verilog HDL since it has a textual format for describing digital circuits and systems. Verilog HDL as a programming language allows a designer to describe a digital circuit in a text format rather than in a schematic format. In fact, Verilog HDL can be used for verification through simulation, timing analysis, test analysis, and logic synthesis.

Verilog HDL can realize digital circuit design and simulate models at algorithmic, RTL, gate, and switch levels. It also provides behavioral and structural synthesis. Here, we simulate models at gate level with structural language. It is defined as the IEEE Standard 1364.

To implement hardware, the design process always follows the top-down design methodology. It takes a designer from abstract concept down to actual hardware step by step. The design flow has to guarantee that the desired functionality is achieved completely and correctly.

Generally, we represent designs in Verilog HDL at three levels of abstraction: behavioral level, register transfer level, and structural level.

In the behavioral level, the design is implemented in terms of the desired algorithm without regards to actual hardware. So it cannot be synthesized into hardware by the synthesis tools.

Design in register transfer level is implicitly realized in terms of hardware registers and combinational logic. The key difference with the behavioral level is that an RTL description can be translated into hardware by the synthesis tools.

Basically, a design in structural level is modeled by explicit description of logic primitives and interconnects between them, and it is also named as a gate-level model.

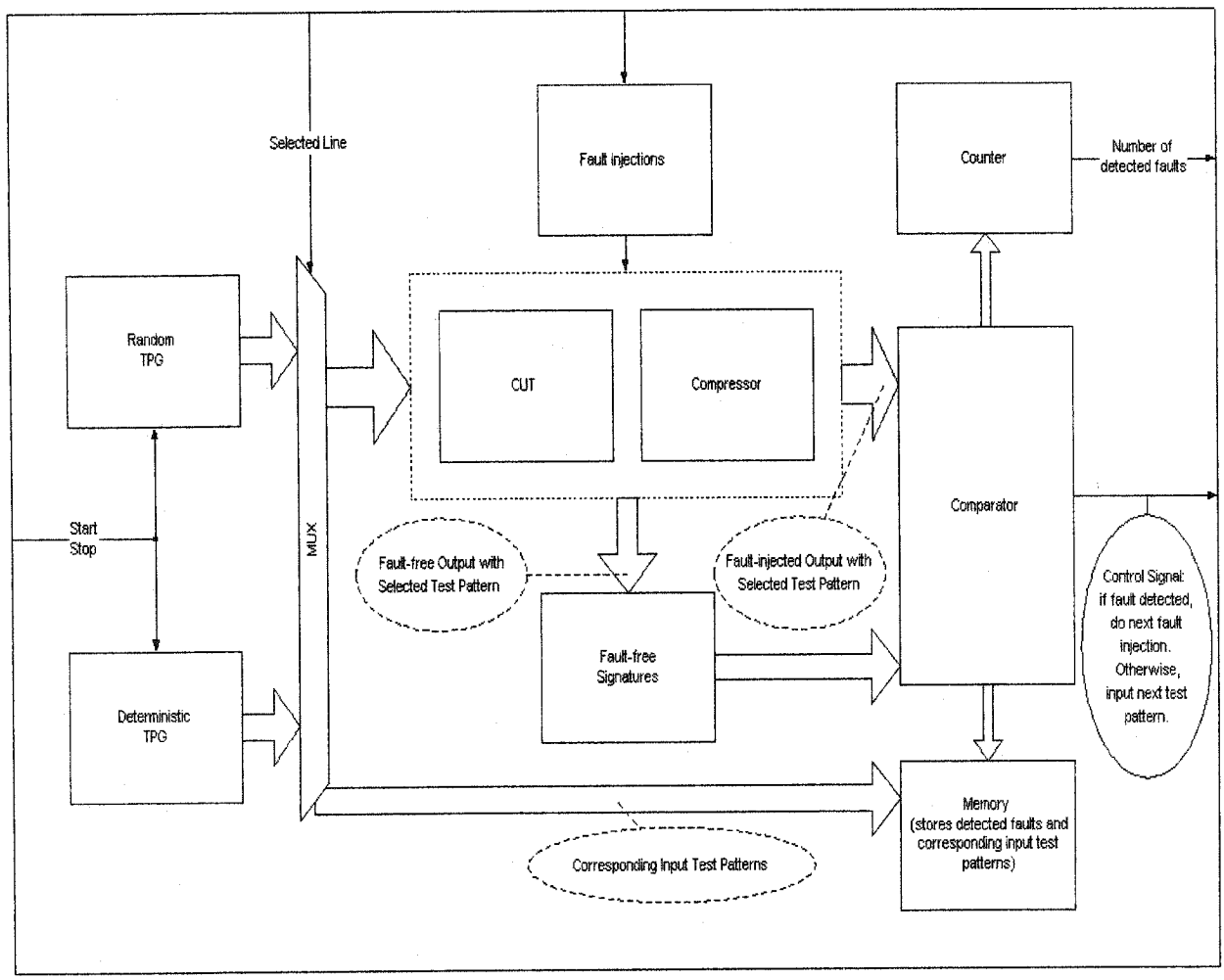

Figure 12. The detailed hardware verification scheme

2. MAX PLUS II DESIGN ENVIRONMENT

Altera MAX PLUS II is an integrated digital circuit simulation software. It consists of 11 application programs and the MAX PLUS II manager. There are hierarchy display, graphic editor, symbol editor, text editor, waveform editor, floorplan editor, compiler, simulator, timing analyzer, programmer, and message processor. Several different design applications can be active simultaneously. Meanwhile, other applications can be run on the background.

Among them, we mainly focus on compiler and simulator applications. Compiler application processes logic projects and perform most tasks automatically. Moreover, the compilation process can be executed for all functions or some functions. Simulator application includes functional simulation, timing simulation, and linked multi-device simulation. Here, we will execute functional simulation.

The diagram in Figure 13 illustrates the design flow process of establishing a new project from conception to completion.

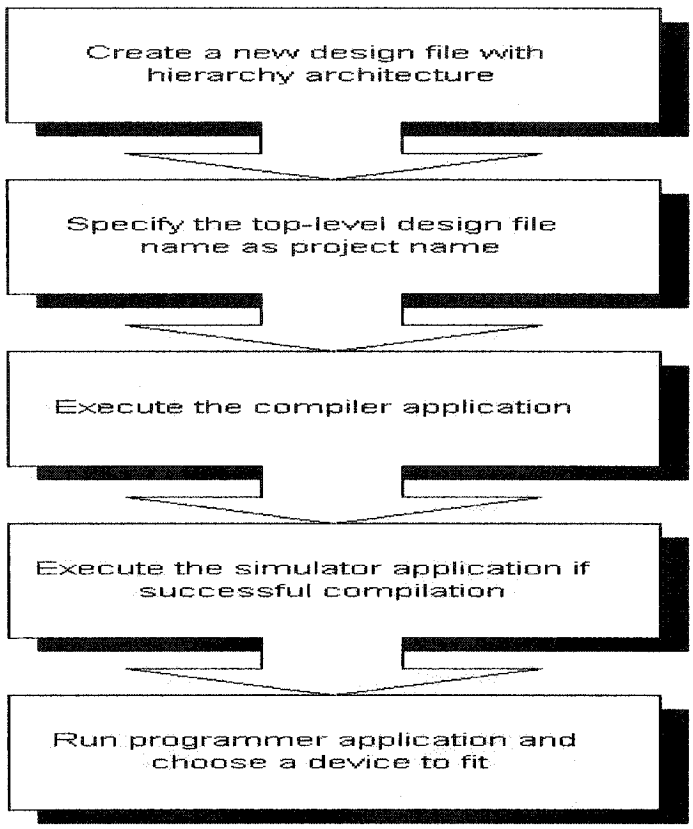

Figure 13. Design flow chart

The MAX PLUS II includes design files, ancillary files, and projects. Design files are used to write ISCAS 85 benchmark circuits with Verilog HDL (suffix is .v). MAX PLUS II compiler can automatically handle graphic, text, or waveform format design files. Ancillary files are associated with design files but do not belong to part of the project hierarchy tree. Here, we import ASCII vector files as input files, and create ASCII table files as output files.

The MAX PLUS II compiler combines with several modules and utilities, such as compiler netlist extractor, database builder, logic synthesizer, partitioner, etc. To improve the efficiency of compiler, we simplify this process and set it for functional SNF extractor compilation. That means this process only includes three modules (compiler netlist extractor, database builder, and functional SNF extractor). It skips the fitter, timing and assembler modules, and facilitates the process. Since the functional SNF does not

contain timing information, it is generated very fast. But the result is created only if a project compiles without any errors.

The MAX PLUS II simulator verifies the logical operation and internal timing of a project. Before simulating a project, we must compile it and get a simulator netlist file (SNF) for functional simulation. During the functional simulation, the simulator ignores any timing delays.

The hardware software co-design test environment enables the testing to be flexible and efficient since the program code is compact to be embeded into a board or a chip. The general flow chart of a testing environment is shown below.

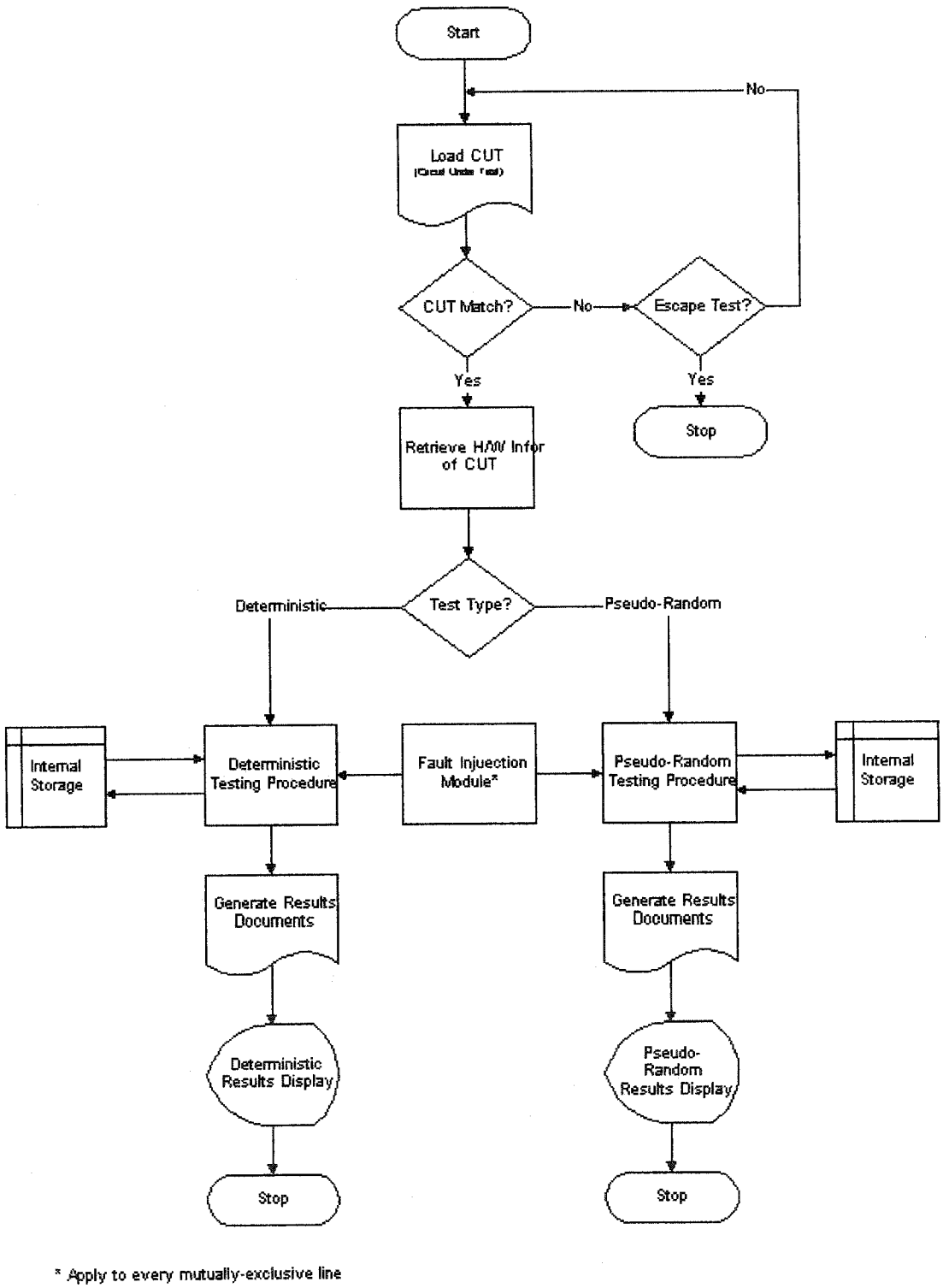

Figure 14. Hardware and software co-design overview

3. TEST PATTERN GENERATION

In our BIST structure, test generation consists of pseudorandom test pattern generator and deterministic test pattern generator. We can make a decision as regards which one is to be used as our test source.

To implement pseudorandom test pattern generator, a simple but efficient algorithm is developed. The program generates vector file as the input of the MUT. A vector file of Altera MAX PLUS II contains information of logic levels of nets within a digital circuit. The simulator uses vectors to simulate the behaviors of the circuit. Vectors for simulation can be defined in vector files (.vec) or simulation channel files (.scf). To realize automatic input file loading, the vector files will be better to simulate, since it is easy to be generated with text file format by the program. One sample vector file for benchmark circuit C17 (C17.vec) is showed as follows:

INTERVAL 1;
START 0;
STOP 3ns;
INPUTS N1 N2 N3 N6 N7;
PATTERN
0>1 0 0 0 1;
1>0 1 0 1 1;
2>1 0 1 0 0;

OUTPUTS N22;

OUTPUTS N23;

Here, "INTERVAL 1; START 0; STOP 3ns;" specifies the time units used. It means the simulation time start from 0 and stop at 3 ns. The interval time is 1. The optional unit section includes the keyword UNIT followed by a time unit. If no time unit is entered, ns is the default unit throughout the vector file. The unit section ends with a semicolon (;).

"INPUTS N1 N2 N3 N6 N7;" means there are five inputs which are N1, N2, N3, N6, and N7. "OUTPUTS N22; OUTPUTS N23;" refers to two outputs in the circuit.

"PATTERN 0>1 0 0 0 1; 1>0 1 0 1 1; 2>1 0 1 0 0;" represents three sequences of pseudorandom numbers generated by a pseudorandom test pattern generator (PTPG)

module. The algorithm [43] for pseudorandom test pattern generator (PTPG) module is as follows:

Algorithm 1

//pseudorandom test pattern generator (PTPG) module is to generate a sequence of pseudorandom numbers that depend on the number of inputs of the specific circuit module. Given an arbitrarily number, we call it "seed number", and it will generate different pseudorandom numbers.

//choose a pair of integers (m and a) as the multiplier and modulo, respectively. Normally, the integer m takes on value 2^{31}-1 (2147483647) and the integer a takes on value 7^5 (16807).

```
//module for pseudorandom test pattern generator (PTPG)
PTPG module
{
        Config modulo as a constant unsigned long integer 2147483647 (2^31-1);
        Config multiplier as a constant unsigned long integer 16807 (7^5);
        Get seed number;
        Config ran as a static unsigned long integer;
        Designate the random number as ran;
        Store ran;

        ran = (ran * multiplier) % modulo;
}

//
main module
{
        //call PTPG module every time
        random_number = PTPG(seed) % m;
}
```

This algorithm generates pseudorandom numbers from 0 to m-1. Here, we set m equal 2. So the pseudorandom numbers are from 0 to 1.

Through this algorithm, we can obtain any pseudorandom number group, which depends on the number of input or combination of input numbers. Then the program generates a vector file or vector files as the input of MAX PLUS II stimulation.

As for the deterministic test pattern generator, it is simpler than pseudorandom test pattern generator, since it is just a textual file followed MAX PLUS II table file format. It is imported to MAX PLUS II as an input file automatically.

For hardware implementation, we can invoke a simple MUX (multiplexer) to implement the choice between pseudorandom test pattern generator and deterministic pattern generator. The following diagram displays the TPG flow chart.

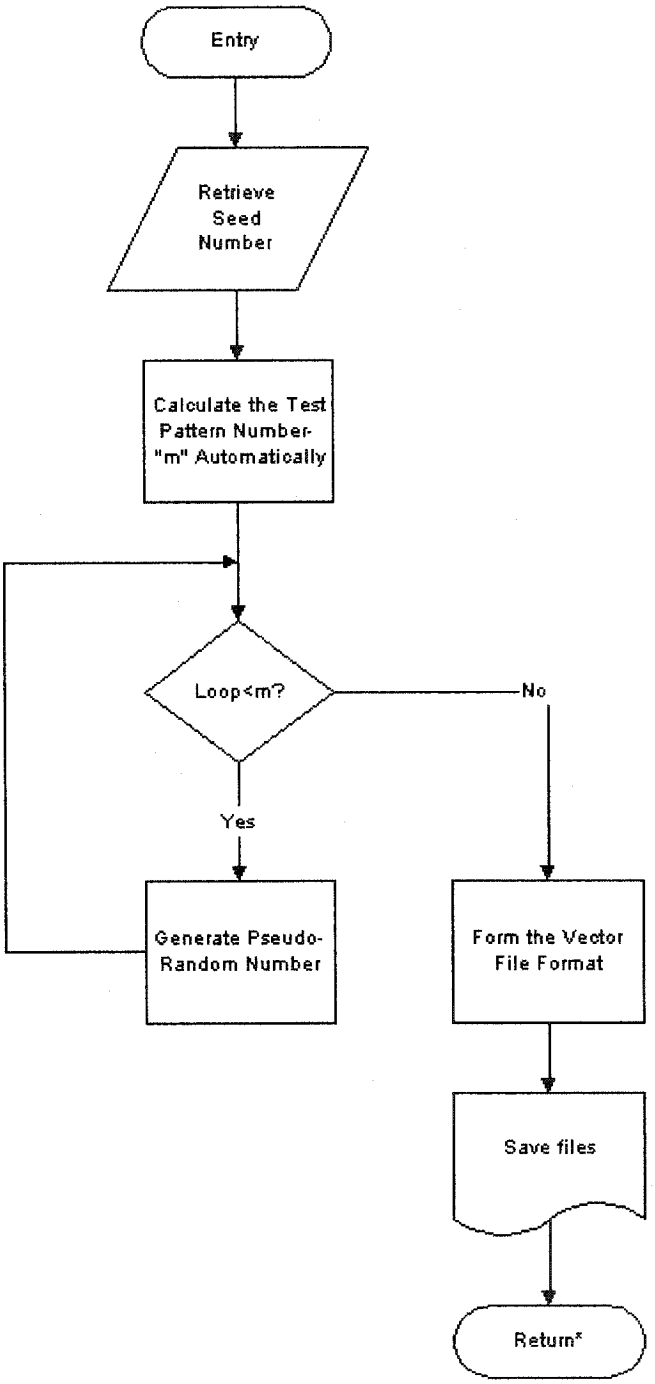

Figure 15. Test pattern generator design flow chart

4. MODULE UNDER TEST

The testing environment is based on ISCAS 85 combinational benchmark circuits. These circuits or MUT (module under test) are written in Verilog HDL.

The MAX PLUS II compiler can import input files with many different formats. It consists of graphic design files (.gdf), text design files (.tdf), VHDL design files (.vhd), Verilog design files (.v), waveform design files (.wdf), Altera design files (.adf), etc. In this work, these circuits for testing are all written with Verilog language.

ISCAS 85 combinational benchmark circuits contain 11 circuits. To realize the simulation, gate-level description includes detailed ports information. So the circuit described with gate-level Verilog is easy to implement with fault injection. In order to accelerate the speed of simulation, we attach some annotation that will not influence the structure of benchmark circuits. The circuit diagram and Verilog HDL description of circuit C17 is given below.

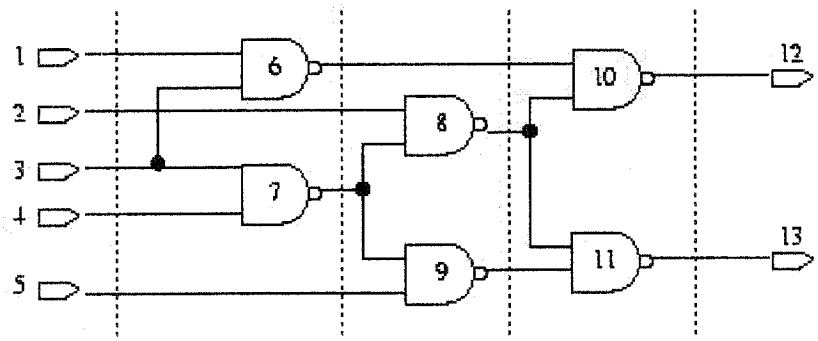

Figure 16. C17 circuit diagram

```
// Verilog
// C17
// Ninputs 5
// Noutputs 2
// Wire 4
// NtotalGates 6
// NAND2 6

module C17 (N1,N2,N3,N6,N7,N22,N23);
```

```
input N1,N2,N3,N6,N7;

output N22,N23;

wire N10,N11,N16,N19;

nand NAND2_1 (N10,N1,N3);
nand NAND2_2 (N11,N3,N6);
nand NAND2_3 (N16,N2,N11);
nand NAND2_4 (N19,N11,N7);
nand NAND2_5 (N22,N10,N16);
nand NAND2_6 (N23,N16,N19);

endmodule
```

Since we have vector file as input, MAX PLUS II can invoke it to do fault-free simulation. The MAX PLUS II will compile this circuit at first. If no errors exist, it will do simulation next. During this procedure, the program will generate a series of intermediate and final files. Among these files, we mainly focus on an output file - table file. The table file format of MAX PLUS II is as follows:

```
%
MAX+plus II 10.1   Date: 08/07/2003 12:46:21
File generated from: C17.scf

INPUTS N1 N2 N3 N6 N7 ;
OUTPUTS N22 N23 ;
UNIT ns ;
RADIX HEX ;
PATTERN
%          N N %
%    N N N N N   2 2 %
%    1 2 3 6 7   2 3 %

 0.0> 0 1 1 1 0 = 0 0
 1.0> 0 0 0 1 0 = 0 0
 2.0> 0 1 1 1 0 = 0 0
 3.0> 0 0 0 0 1 = 0 1
 4.0> 0 1 0 1 0 = 1 1
 5.0> 1 0 1 0 1 = 1 1
 6.0> 1 0 0 0 0 = 0 0
 7.0> 0 0 0 1 0 = 0 0
 8.0> 1 0 1 0 1 = 1 1
 9.0> 1 0 0 0 0 = 0 0
10.0> 1 1 1 1 0 = 1 0
```

```
11.0> 1 1 1 0 1 = 1 1
12.0> 0 1 1 0 1 = 1 1
13.0> 0 1 0 1 1 = 1 1
14.0> 0 1 0 1 0 = 1 1
15.0> 1 1 1 1 1 = 1 0
16.0> 1 1 0 1 1 = 1 1
17.0> 0 0 0 0 0 = 0 0
18.0> 1 0 0 0 0 = 0 0
19.0> 0 1 1 0 1 = 1 1
20.0> 1 0 0 1 0 = 0 0
21.0> 0 0 0 1 1 = 0 1
22.0> X X X X X = X X
;
```

In this table file, we will ignore the description part and mainly focus on the values of inputs (N1, N2, N3, N6, N7) and outputs (N22, N23). From "0.0>", we have five input values and two output values separated by equal symbol. In fact, these values accurately match input pins N1 to N7 and output pins N22 to N23.

The MUT includes a series of benchmark circuits described with gate-level Verilog language.

Algorithm 2

//The program will automatically pick up input values and output values, and store to memory. The algorithm for the MUT is illustrated as follows:

//Opening Verilog file of the MUT and generating the vector file for the MUT.
Find all relevant information of the MUT, such as numbers and names of all inputs, outputs and wires. Define seed number and test pattern number for specific testing.

//Finding the parameters of Verilog file
Finding the parameters of Verilog file for

{
Designate the inputs array as INPUT
Store INPUT;
Store Ninputs;
and

```
        Designate the outputs array as OUTPUT
        Store OUTPUT;
        Store Noutputs;
        and
        Designate the wires array as WIRE
        Store WIRE;
        Store Nwires;
        }

//Determining the parameters of vector file:
Determine the time of STARTs and STOPs of vectors;
Determine the size of INTERVAL of vectors;

//Forming the pseudorandom numbers for defined ports (Algorithm 1):
Designate the PATTERN of vectors and forming the pseudorandom numbers for defined ports
        For (i=0; i<b_test_pattern; i++)
//Find how many test pattern numbers will be applied;
        {
        for (count = 0; count < m_input; count++)
        //Find how many input ports in circuit;
        Generate pseudorandom number in specific port;
        }

//Finding the output vector names
Finding the output vector names and replacing them in the vector file;

//Close vector file
```

Algorithm 3

//The algorithm can automatically conduct MAX PLUS II to run on the background. The algorithm simplifies the compiler and simulation procedure to improve the performance of MAX PLUS II.

```
//Preforming a functional SNF extractor file
Preforming a functional SNF extractor file:
        {
        Open an empty functional SNF extractor file with write mode;
        Append one line
        "COMPILER_PROCESSING_CONFIGURATION\nBEGIN\nFUNCTIONAL_SNF_EX
        TRACTOR = ON;
        END;"
        Close this functional SNF extractor file
        }

//Conducting MAX PLUS II to do compilation and simulation on the background
Conducting MAX PLUS II to do compilation and simulation for
        {
        Do compilation for specific circuit on the background;
        Do simulation for specific circuit with inputting vector file;
        Get table file after successful completion.
        Duplicate the table file for future fault injection;
        }
```

The compressor module can be included into the MUT. Therefore, the output values are different from the former values, which have been compressed by a specific algorithm to reduce the amount of output values storage.

The extra logic of compressor unit must be as simple as possible, and easily embeded within the MUT. Certainly, it would not introduce signal delays and normal functionality of the MUT.

5. Fault Injection and Comparator

Fault injection module is the most important part of the simulation. The hardware fault injection technique is essential in order to iteratively insert stuck-at-0 and stuck-at-1

faults into every mutually exclusive wire. The total number of wires consists of primary inputs, primary outputs, and internal wires. To implement the fault injection sequentially, we consider these three parts separately.

The advantage of separation is that we can invoke different algorithm to achieve high performance. Due to the limited feature of MAX PLUS II, we have to treat the circuit module as a black box. For input and output of specific module after fault injection, the structure of this module could not be changed. We just need to change the real values of input or output. For internal wires of a specific circuit module during injecting one fault at a time, the structure of the circuit have been modified, since faults have been inserted in some part of the circuit.

Fault injection approach is based on inserting faults into module, specifically, through changing the value of a particular net to realize the fault injection in the whole circuit. Since we apply MAX PLUS II as the simulator (we can not change the internal structure of MAX PLUS II), the program runs on the background of MAX PLUS II. So it is completely independent of the simulator. To speed up the simulation, we invoke some methods to improve the performance of the program.

To realize injecting stuck-at-faults, the simulator is modeled by forcing the value of the subject node to all ones or all zeros. It goes through all nodes in the circuit iteratively with stuck-at-1 and stuck-at-0. In fact, we have to break a wire where we tend to inject a fault. We do not concern about the value of forepart of the wire and put stuck-at signal value straight to the rear part of the wire.

Since fault injection in different part of the circuit module needs different algorithms, it is better to separate them to speed up the running of the program. The following flow chart shows the fault injection in input ports.

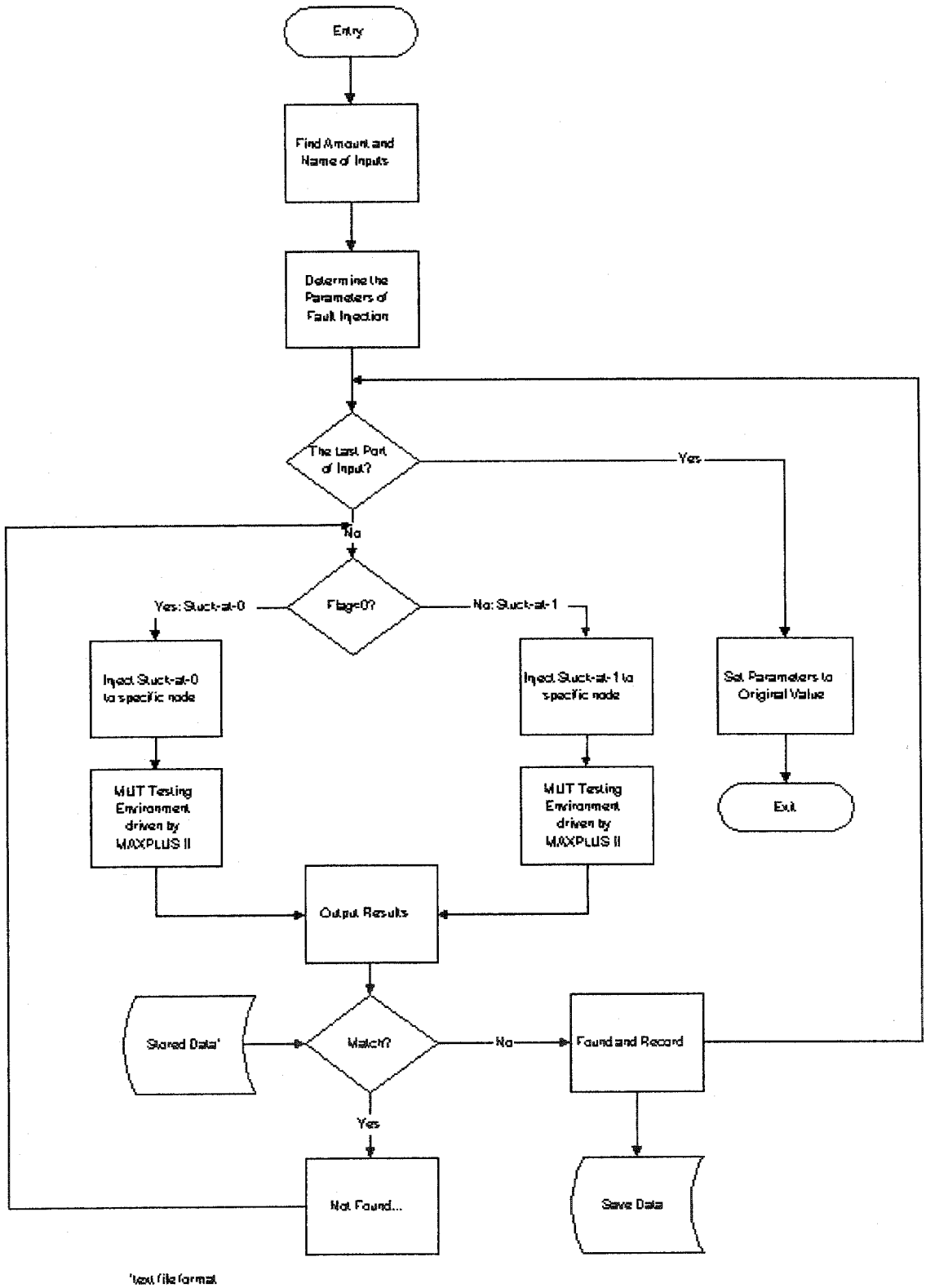

Figure 17. Fault injection in input part

The fault injection scheme in output ports is similar to that for the input. The difference is that we skip the MUT testing procedure driven by MAX PLUS II. The advantage of this approach is obviously that it improves the efficiency of program running. The flow chart of fault injection in output is shown as follows.

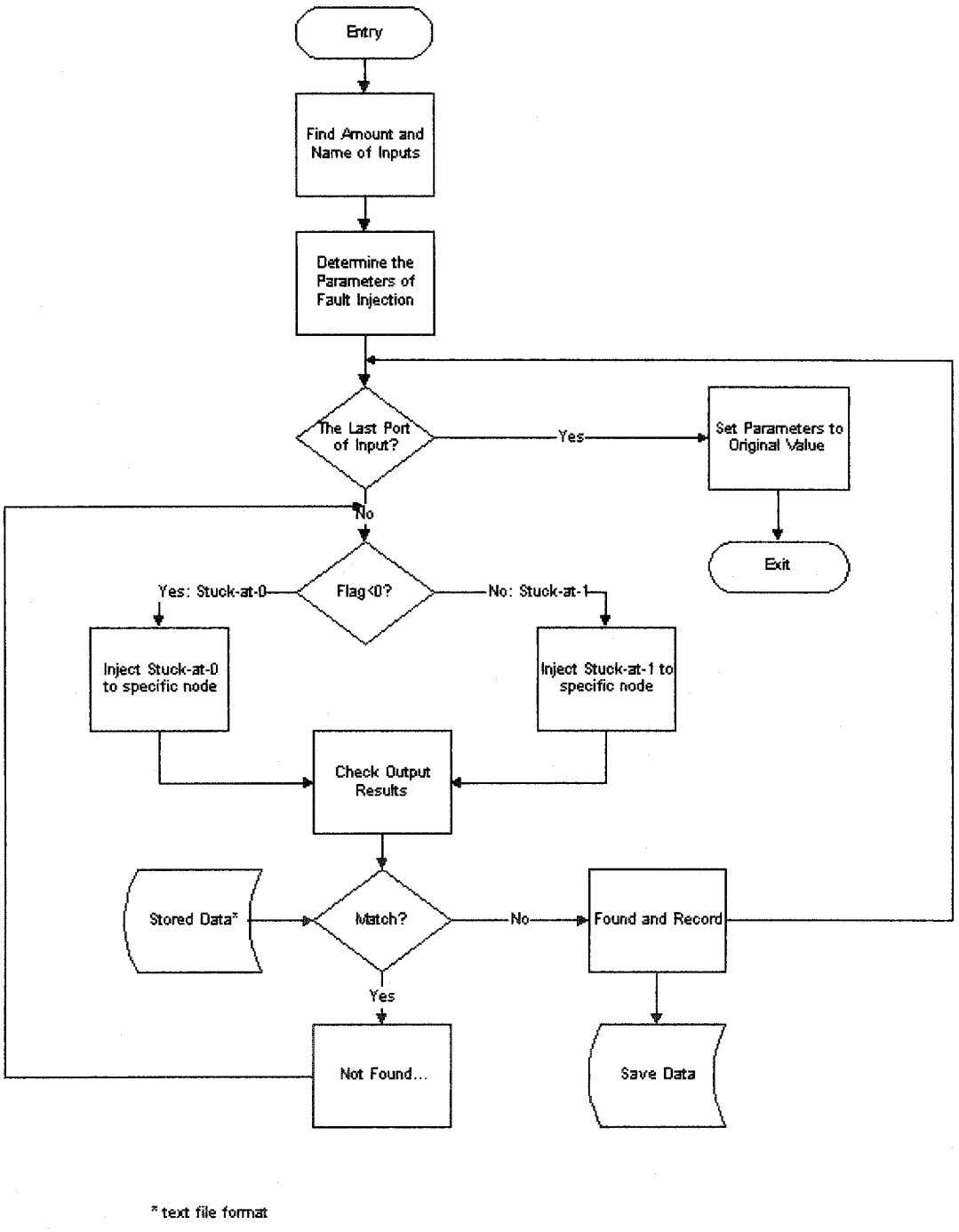

Figure 18. Fault injection in output part

The fault injection in wires is quite difficult to implement due to several reasons. The first is that the circuit structure changes when the fault is injected into the circuit. We have to compile and simulate the circuit every time since the circuit is to be treated as a new one. Besides, we have to identify the wire. Due to huge number of wires in a large circuit, it is a challenge to speed up the program running. We need an optimized algorithm to enhance efficiency. The fault injection scheme in wire part is displayed as follows.

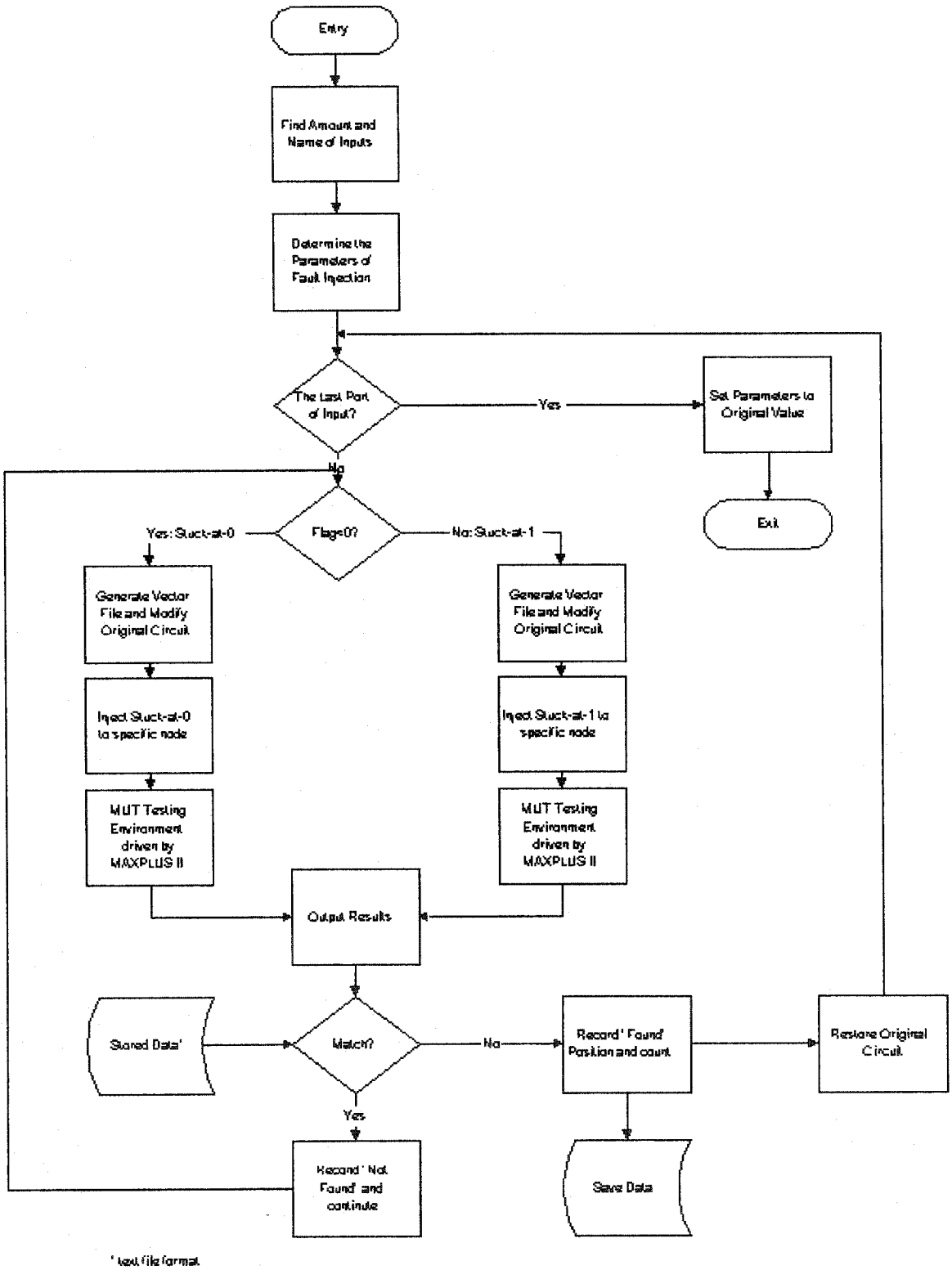

Figure 19. Fault injection in wire part

Algorithm 4

//The pseudocode description of fault injection (input pins) algorithm that can insert faults in input ports iteratively with stuck-at-0 and stuck-at-1 faults.

```
//Finding number and name of inputs:
Search the circuit (Verilog format) that is to be tested and find number and name of inputs. Store
in memory (Algorithm 2)

//Determining the parameters of fault injection:
Determining the symbol bit "flag" of stuck-at-0 or stuck-at-1;
Determining the sequence number of test pattern that was applied;
Determining the test pattern number where the fault has been found;

//Injecting one fault at a time in the test circuit
while (start from the first input port, go next by next until the last port)
        {
        //Determining fault type (stuck-at-0 or stuck-at-1)
        if (flag is less than 0)
                symbol flag is stuck-at-0;
        else
                symbol flag is stuck-at-1;

//Taking the test pattern to generate vector file and get output table file (Algorithm 3)
        while (invoking test pattern line by line with vector file until the last line)
                {
                if (output value matches)
                        no fault found;
                else
                        fault found;
                        Sequence number increases by one;
                }
        If (symbol flag is less than 0) switch to stuck-at-1;
```

Set up the symbol flag;

Clear sequence number to 0;

}

Algorithm 5

//The pseudocode description of fault injection (output pins) algorithm that can insert faults in output ports iteratively with stuck-at-0 and stuck-at-1 faults.

//Finding number and name of outputs:
Search the circuit (VERILOG format) that is to be tested and find number and name of outputs. Store in memory (Algorithm 2)

//Determining the parameters of fault injection:
Determining the symbol bit "flag" of stuck-at-0 or stuck-at-1;
Determining the sequence number of test pattern that was applied;
Determining the test pattern number where the fault has been found;

//Injecting one fault at a time on the test circuit
while (start from the first output port, go next by next until the last port)

```
{
//Determining fault type (stuck-at-0 or stuck-at-1)
if (flag is less than 0)
        symbol flag is stuck-at-0;
else
        symbol flag is stuck-at-1;
```

while (invoking test pattern line by line with vector file until the last line)

```
{
if (output value matches)
        no fault found;
else
```

fault found;

Sequence number increases by one;

}

If (symbol flag is less than 0) switch to stuck-at-1;

Set up the symbol flag;

Clear sequence number to 0;

}

Algorithm 6 (wire)

//The pseudocode description of fault injection (internal wires) algorithm that can insert faults in internal wires iteratively with stuck-at-0 and stuck-at-1 faults.

//Finding number and name of mutually exclusive wires:
Search the circuit (VERILOG format) that is to be tested and find number and name of internal wires. Store in memory (Algorithm 2)

//Determining the parameters of fault injection:
Determining the symbol bit "flag" of stuck-at-0 or stuck-at-1;
Determining the sequence number of test pattern that was applied;
Determining the test pattern number where the fault has been found;

//Injecting one fault at a time on the testing circuit
while (start from the first mutually-exclusive wire, go next by next until the last wire)

{

//Determining fault type (stuck-at-0 or stuck-at-1)

if (flag is less than 0)

Symbol flag is stuck-at-0;

else

Symbol flag is stuck-at-1;

```
//searching for the name of wire in circuit and store to {w1, w2}
if (found)
        //inject fault in specific area
        Replace it with symbol flag;
else
continue to search;

//Taking the test pattern to generate vector file and get output table file (Algorithm 3)
while (invoking test pattern line by line with vector file until the last line)
        {
        if (output value matches)
                no fault found;
        else
                fault found;
                Sequence number increases by one;
        }

If (symbol flag is less than 0) switch to stuck-at-1;

Set up the symbol flag;

Clear sequence number to 0;
}
```

The sequence number indicates which test pattern has been applied. If the fault has not been detected, the program will take next test pattern to run until the fault can be detected. In this case, the sequence number will be recorded and will indicate which test pattern can detect this fault.

The symbol flag refers to which fault has been inserted (stuck-at-0 or stuck-at-1). Since the procedure is for a specific wire (including input and output), the faults (stuck-at-0 or stuck-at-1) are injected iteratively, going on to the next wire until all wires have been tested. In program, the symbol indicates the fault type.

After applying a set of vector files, the circuit with Verilog language is evaluated. The results are saved in the file 'circuit_name.log'.

6. OTHER COMPONENTS

Counter module provides functions to record the number of detected faults for every test pattern. Since the test patterns we applied are commonly very huge for large circuits, it is necessary to manage this counter group with minimum operation.

The memory module stores detected faults and the corresponding input test patterns. To save this massive information, the storing method is commonly adopted as file format to save computer memory. The memory module stores all information about testing that includes applied test patterns, detected faults, and where to find them. The result is yielded from comparator module through comparison of the outputs with fault-free outputs coming from fault-free module.

Fault-free signatures come from the MUT and Compressor modules without any faults injected. Fault-free signatures correspond to relative test patterns. Comparator module can utilize this module to compare the outputs from the MUT and Compressor module with fault injection.

7. SYNTHESIS

After testing every module separately to ensure its function, we put these modules together. These modules can be synthesized as a practical simulator. To test a logic circuit (gate-level Verilog code), we invoke pseudorandom TPG or deterministic TPG to generate the result and log file finally. During this procedure, MAX PLUS II is run on the background.

In order to simulate a given circuit, we shall put this file (circuit described with Verilog code) with simulator software under a same folder. Since the simulator program will call

MAX PLUS II during program running, the direction (path) to MAX PLUS II has to be indicated at first. It means the user should set up the path manually. The reason is that the simulator software must adapt to all kinds of running environments. But normally the path of MAX PLUS II is different.

The simulator program not only understands the ISCAS 85 benchmark circuits, but also all kinds of circuits with Verilog code (gate-level description). To achieve this goal, the program should be flexible enough for adaptation during simulation.

CHAPTER 4

Experimental Results

1. FAULT COVERAGE

The proposed fault simulator was implemented on an IBM-compliant personal computer with Pentium III 600, and 128 MB RAM. To demonstrate the feasibility of the proposed approaches, independent simulations were conduced on various ISCAS 85 combinational benchmark circuits.

Some of the major objectives achieved through the research could be summarized as follows:

- Developed a complete fault simulator using C language as general parser, with a circuit Text description for Verilog conversion;
- Implemented combinational logic circuit test simulation environment;
- Simulated ISCAS 85 combinational benchmark circuits using hardware pseudo-random input test pattern generator and saved deterministic input test vectors;
- Realized the verification scheme and compared the results to theoretical ones, in order to show the validity of the proposed test simulation algorithms;
- Investigated and designed Verilog implementation of the verification scheme; and
- Realized the solution using Altera MAX PLUS II design environment and compared the results to theoretical ones.

A software hardware co-design methodology is to synchronise the on-board processor and configurable logic blocks. The tested circuits were realized using Verilog language. The design has to be configured, along with the pseudorandom input test pattern generator and output registers, and downloaded and verified on an Altera FPGA board.

The logic circuit test simulator we developed has several important advantages as compared to other simulators. Since the proposed simulator is based on software

hardware co-design methodology, obviously it can be invoked for practical testing environments.

The first advantage is that the test circuits can be written in any hardware description language. No matter whether we use Verilog, VHDL, or AHDL to describe the circuits, the simulator can handle them efficiently.

Many simulators written for the benchmark circuits are specifically intended for the netlist formats of the circuits. The circuits in our implementations were written in hardware description language and the developed simulator can be used to test all such circuits described in any hardware description language.

As was previously mentioned, the simulator was developed using C language, with a circuit Text description for Verilog conversion. The simulator is fairly compact to be embeded in a hardware-testing environment since we have realized a complete logic circuit test simulation environment besides our simulator. It means that the design can be configured, along with the pseudorandom input test pattern generator and the output registers, and downloaded and verified on an Altera FPGA board.

The experimental results for ISCAS 85 combinational benchmark circuits are shown in Tables 2 and 3. Table 2 presents experimental for pseudorandom inputs, while Table 3 provides results for ISCAS 85 benchmark circuits using deterministic compacted test vectors. Using the fault simulator, we determined the fault coverage and CPU simulation time required for all ISCAS 85 benchmark circuits without using space compactors.

Table 2. Results on ISCAS 85 benchmark circuits - pseudorandom test vectors

Circuit name	No of test vectors	Fault coverage (%)	Total time (secs)	Simulation time (secs)
C17	22	100	20	8
C432	199	95.918	1076	430
C499	199	88.889	1027	411
C880	199	92.551	2875	1150
C1355	199	87.223	3858	1543
C1908	199	83.571	9323	3729
C5315	199	97.988	103899	41559
C6288	199	99.306	148271	59308

Table 3. Results on ISCAS 85 Benchmark Circuits - deterministic test vectors

Circuit name	No of test vectors	Fault coverage (%)	Total time (secs)	Simulation time (secs)
C17	5	100	18	7
C432	46	95.918	542	217
C499	38	88.889	673	269
C880	49	92.551	1923	769
C1355	53	87.223	3363	1345
C1908	45	83.571	8450	3380
C5315	97	97.988	98496	39398
C6288	48	99.306	117088	46835

The following figures show the same results, using both the pseudorandom and deterministic test patterns, but in a graphical way.

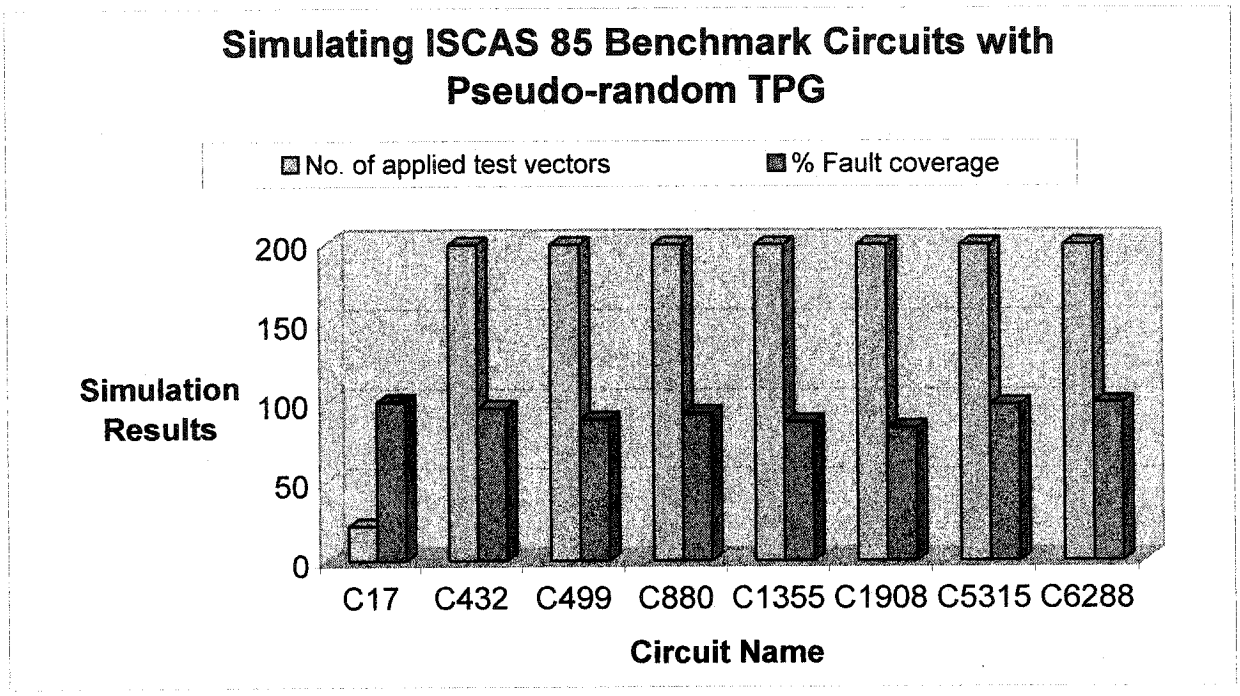

Figure 20. Simulating ISCAS 85 benchmark circuits with pseudo-random TPG

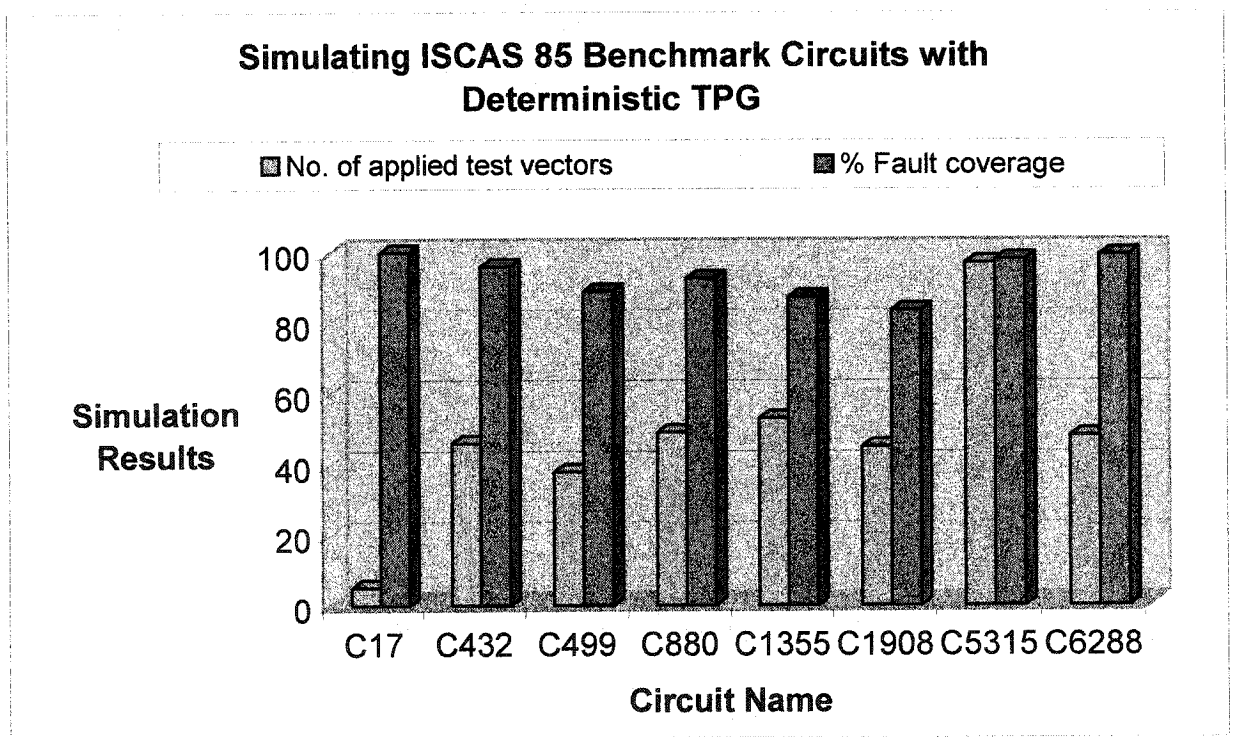

Figure 21. Simulating ISCAS 85 benchmark circuits with deterministic TPG

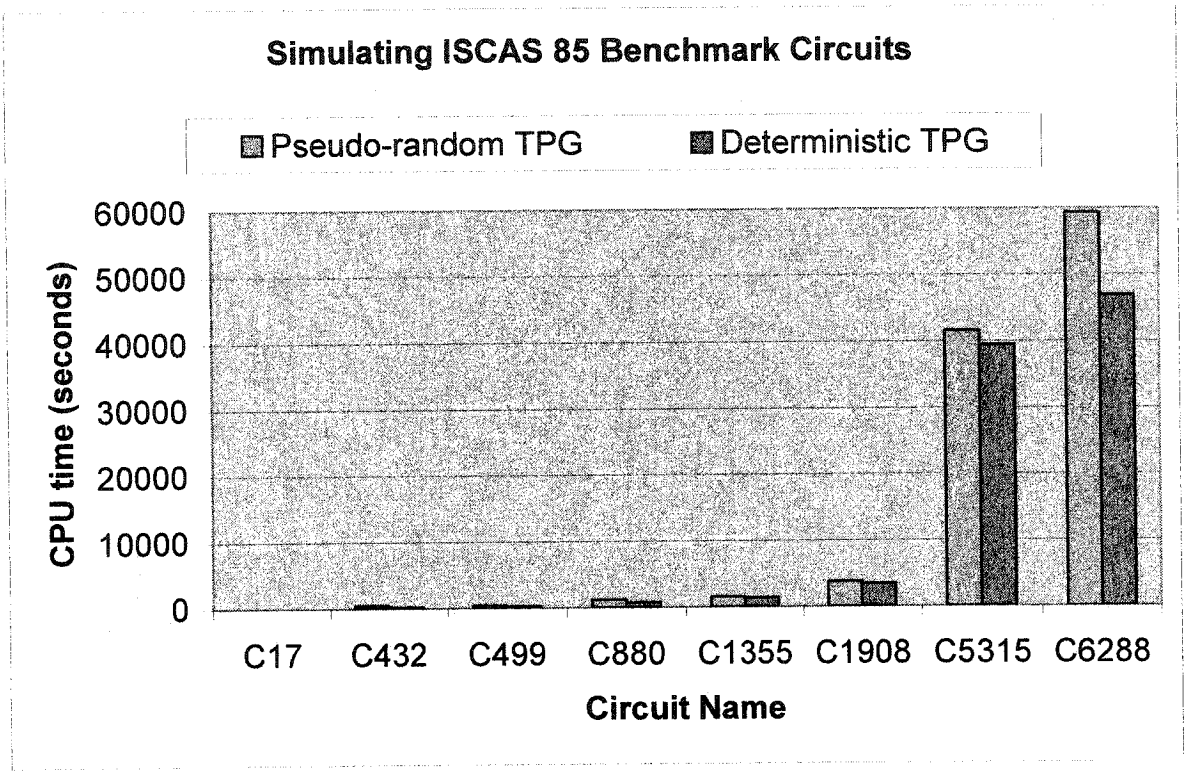

Figure 22. Simulating ISCAS 85 benchmark circuits

2. ANALYSIS

The results file "circuit_name.log" contains information from the fault simulation. It is a text format file, which records the detected faults and fault numbers in details. The format of C17.log is as follows:

Log file for the circuit C17.bench.
Number of faults detected by each test pattern:

test 1 :9 faults detected 0 1 1 1 0 = 0 0
test 2 :2 faults detected 0 0 0 1 0 = 0 0
test 3 :0 faults detected 0 1 1 1 0 = 0 0
test 4 :4 faults detected 0 0 0 0 1 = 0 1
test 5 :4 faults detected 0 1 0 1 0 = 1 1
test 6 :3 faults detected 1 0 1 0 1 = 1 1
test 7 :0 faults detected 1 0 0 0 0 = 0 0
test 8 :0 faults detected 0 0 0 1 0 = 0 0
test 9 :0 faults detected 1 0 1 0 1 = 1 1
test 10:0 faults detected 1 0 0 0 0 = 0 0
test 11:0 faults detected 1 1 1 1 0 = 1 0

test 12:0 faults detected 1 1 1 0 1 = 1 1
test 13:0 faults detected 0 1 1 0 1 = 1 1
test 14:0 faults detected 0 1 0 1 1 = 1 1
test 15:0 faults detected 0 1 0 1 0 = 1 1
test 16:0 faults detected 1 1 1 1 1 = 1 0
test 17:0 faults detected 1 1 0 1 1 = 1 1
test 18:0 faults detected 0 0 0 0 0 = 0 0
test 19:0 faults detected 1 0 0 0 0 = 0 0
test 20:0 faults detected 0 1 1 0 1 = 1 1
test 21:0 faults detected 1 0 0 1 0 = 0 0
test 22:0 faults detected 0 0 0 1 1 = 0 1

End of fault simulation.

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

1.	*Circuit Structure*		
	Name of the circuit	:	*C17*
	Number of gates	:	*11*
	Number of primary inputs	:	*5*
	Number of primary outputs	:	*2*
	Depth of the circuit	:	*4*
2.	*Simulation parameters*		
	Simulation mode	:	*random*
	Initial random number generator seed	:	*1050420308*
3.	*Simulation results*		
	Number of test patterns applied	:	*22*
	Fault coverage	:	*100.000 %*
	Number of collapsed faults	:	*22*
	Number of detected faults	:	*22*
	Number of undetected faults	:	*0*
4.	*Total running time*	:	*20 seconds*
	Total Simulation time	:	*8 seconds*

** End*

Based on our conducted simulation for some combinational circuits, we obtained the above log file. In this log file, we can find all information we need. For each circuit, we determined the number of test vectors used, CPU simulation time, and percentage of fault coverage by running the simulator on a PC-compatible workstation.

CHAPTER 5

Conclusion and Future Work

In this thesis, we developed fault simulator for combinational circuits based on Altera MAX PLUS II development environment, and provided experimental results on ISCAS 85 combinational benchmark circuits utilizing the simulator based on the use of deterministic compact and pseudorandom test inputs. The simulation results validate the efficacy of the developed approaches.

However, there remains unfinished works to make system complete and comprehensive as a fault simulation system, capable of handling other faults models than stuck-at-faults. We believe that the system can be used to handle most classes of logic circuits described in Verilog HDL. Besides, the simulation technique seems suitable for collecting statistical data on logic circuits as well.

The simulator can be adapted to handle logic circuits specified in other kinds of hardware description languages, such as VHDL, etc. The system can automatically detect which HDL the testing file uses and then treat it accordingly.

To deal with sequential logic, we just need to change formats in test pattern files. For example, due to the existence of flip-flops in the circuits, the sequential logic circuits need several time clocks to reach stable states. We can append some signals (or reset signals) in test pattern file to handle this problem.

Further studies will be done in relation to embeded cores-based systems, specially targeted towards IP (intellectual property) cores. In the latest SOC (system-on-a-chip) design, many independent modules can be contained in a single IP core. IP cores can be CPUs, memories, DSPs, and other various kinds of communication modules. The advantage of using IP cores [58] is that they speed up the design cycle of large complex systems, and thereby achieve a shorter time-to-market. But it also increases challenge for test engineers.

REFERENCES

1. S. R. Das, E. M. Petriu, T. Barakat, M. H. Assaf and A. R. Nayak, "Space compaction under generalized mergeability, IEEE Transactions on Instrumentation and Measurement", vol. 47, pp. 1283-1293, October 1998

2. K. Chakrabarty, Ph.D. Dissertation, Department of Computer Science and Engineering, University of Michigan, "Test Response Compaction for Built-In Self-Testing". Ann Arbor, MI, 1995

3. H. K. Lee and D. S. Ha, Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA, "On the generation of test patterns for combinational circuits", Technical Report 12-93, 1993

4. S. R. Das, M. H. Assaf, E. M. Petriu, W. B. Jone and K. Chakrabarty, "A novel approach to designing aliasing-free space compactors based on switching theory formulation," Proc. IEEE Instr. Meas. Tech. Conf., pp.pp.198-203, 2001

5. J Biggs, ARM Ltd. A Gibbons, Synopsys Inc. "Reference Methodology for Enabling Core Based Design", European Synopsys User Group, March 2002

6. Z Peng, Embeded Systems Lab. P Eles, E Larsson, G Jervan, A R. Mohamed, Y Sun and G Carlsson, Ericsson Radio Systems. "SoC Design for Testability". 2001-07-01

7. Neil H.E. Weste, Kamran Eshraghian, "PRINCIPLES OF CMOS VLSI DESIGN", second edition, Addison-wesley publishing company

8. M. Michael Vai, "VLSI Design", CRC Press. 2001

9. Parag K. Lala, Electrical Engineering Department, North Carolina Agricultural and Technical State University, "Digital Circuit Testing and Testability", Academic Press. 1997

10. Paul H. Bardell, William H. McAnney, Jacob Savir, International Business Machines Corporation, Poughkeepsie, New York, "'Built-In Test for VLSI: Pseudorandom Techniques", 1987, John Wiley & Sons, Inc.

11. M. C. Hansen and J. P. Hayes, "High-level test generation using physically-induced faults", *Proc. 1995 VLSI Test Symp.* pp. 20-28, 1995

12. N. K. Jha, Electrical Engineering at Princeton University and head of the Center of Embeded System-on-a-chip Design, Sandeep Gupta, Electrical Engineering Department, University of Southern California, "Testing of Digital Systems", Cambridge University press. 2003

13. V. N. Yarmolik, Minsk Radio Engineering Institute, USSR, "Fault Diagnosis of Digital Circuits", John Wiley & Sons. 1990

14. T. W. Williams, IBM Corporation, General Technical Division, Bourlder, CO 80302, U.S.A. "VLSI Testing", Elsevier Science Publishers B.V., 1986

15. K. Chakrabarty, M. G. Karpovsky and L. B. Levitin, "Fault isolation and diagnosis in multiprocessor systems with point-to-point connections", In Fault-Tolerant Parallel and Distributed Systems, D. R. Avresky and D. R. Kaeli, (eds.), pp. 285-301, Kluwer Academic Publishers, Norwell, MA, 1998

16. S. R. Das, T. F. Barakat, E. M. Petriu, M. H. Assaf and K. Chakrabarty, "Space compression revisited", IEEE Transactions on Instrumentation and Measurement, vol. 49, pp. 671-678, June 2000

17. K. Chakrabarty, "Optimal test access architectures for system-on-a-chip", ACM Transactions on Design Automation of Electronic systems, vol. 6, pp. 26-49, January 2001

18. S. R. Das, J. Y. Liang, E. M. Petriu, M. H. Assaf, W. -B. Jone and K. Chakrabarty, "Data compression in space under generalized mergeability based on concepts of cover table and frequency ordering", IEEE Transactions on Instrumentation and Measurement, vol. 51, pp. 150-172, February 2002

19. K. Chakrabarty and B. T. Murray, "Design of built-in test generator circuits using width compression", IEEE Transactions on CAD/ICAS, vol. 17, pp. 1044-1051, October 1998

20. T. Zhang, F. Cao, A. Dewey, R. B. Fair and K. Chakrabarty, "Performance analysis of microelectrofluidic systems using hierarchical modeling and simulation", IEEE Transactions on Circuits and Systems (Part II: Analog and Digital Signal Processing), vol. 48, pp. 482-491, May 2001

21. A. Chandra and K. Chakrabarty, "System-on-a-chip test data compression and decompression based on internal scan chains and Golomb coding", IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems, vol. 21, pp. 715-722, June 2002

22. V. Ivengar, K. Chakrabarty and E. J. Marinissen, "Efficient test access mechanism optimization for system-on-a-chip", IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems, vol. 22, pp. 593-604, May 2003

23. C. Liu and K. Chakrabarty, "Failing vector identification based on overlapping intervals of test vectors in a scan-BIST environment", IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems, vol. 22, pp. 593-604, May 2003

24. V. Iyengar, K. Chakrabarty and E. J. Marinissen, "Test access mechanism optimization, test scheduling and tester data volume reduction for system-on-a-chip", accepted for publication in IEEE Transactions on Computers, 2003

25. P. F. Flores, H. C. Neto, K. Chakrabarty and J. P. M. Silva, "Test pattern generation for width compression in BIST", Proc. 1999 IEEE International Symposium on Circuits and Systems, pp. 114-118, Orlando, FL, May 1999

26. K. Chakrabarty, "Design of system-on-a-chip test access architectures using integer linear programming", Proc. 2000 IEEE VLSI Test Symposium, pp. 127-134, 2000

27. T. Zhang, K. Chakrabarty and R. B. Fair, "Design of reconfigurable composite Microsystems based on hardware/software co-design principles", Proc. International Conference on Modeling and Simulation of Microsystems, pp. 148-152, 2001

28. E. J. Marinissen, V. Iyengar, K. Chakrabarty, "A set of benchmarks for modular testing of SOCs", Proc. IEEE International Test Conference, pp. 519-528, 2002

29. V. Iyengar, K. Chakrabarty, E. J. Marinissen, "Recent advances in TAM optimization, test scheduling, and test resource management for modular testing of core-based SOCs", Proc. IEEE Asian Test Symposium, pp. 320-325, 2002

30. D. K. Pradhan, C. Liu and K. Chakrabarty, "EBIST: A novel test generator with built-in fault detection capability", Proc. IEEE/ACM Design, Automation and Test in Europe (DATE) Conference, pp. 220-224, 2003

31. K. Chakrabarty, B. T. Murray and J. P. Hayes, "Optimal zero-aliasing space compaction of test responses", IEEE Transactions on Computers, vol. 47, pp. 1171-1187, November 1998

32. V. Iyengar, K. Chakrabarty and B. T. Murray, "Deterministic built-in pattern generation for sequential circuits", Journal of Electronic Testing: Theory and Applications, vol. 15, pp. 97-114, August/October, 1999

33. V. Iyengar, K. Chakrabarty and B. T. Murray, "Deterministic built-in self testing of sequential circuits using precomputed test patterns", Proc. 1998 IEEE VLSI Test Symposium, pp. 418-423, April 1998

34. K. Chakrabarty, "Test scheduling for core-based systems", Proc. International Conference on Computer-Aided Design (ICCAD), PP. 391-394, 1999

35. M. Seuring and K. Chakrabarty, "Space compaction of test responses for IP cores using orthogonal transmission functions", Proc. IEEE VLSI Test Symposium, pp. 213-219, 2000

36. K. Chakrabarty and S. Swaminathan, "Built-in self testing of high-performance circuits using twisted-ring counters", Proc. 2000 IEEE International Symposium Conference (DAC), pp. I-72-I76, 2000

37. K. Chakrabarty, R. Mukherjee and A. Exnicios, "Synthesis of transparent circuits for hierarchical and system-on-a-chip test", Proc. IEEE International Conference on VLSI Design, PP. 431-436, Bangalore, India, January 2001

38. B.B. Bhattacharya, A. Dmitriev, M. Goessel and K. Chakrabarty, "Synthesis of single-output space compactors with application to scan-based IP cores", Proc. Asia South Pacific Design Automation Conference, pp. 496-501, 2001

39. S. Swaminathan and K. Chakrabarty, "A deterministic scan-BIST architecture with application to field testing of high-availability systems", Proc. IEEE Custom integrated Circuits Conference, pp.259-262, May 2001

40. V. Iyengar, K. Chakrabarty and E. J. Marinissen, "Test wrapper and test access mechanism co-optimization for system-on-a-chip", Proc. IEEE International Test Conference, pp. 1023-1032, 2001

41. V. Iyengar, S. K. Goel, E. J. Marinissen and K. Chakrabarty, "Test resource optimization for multi-site testing of SOCs under ATE memory depth constraints", Proc. IEEE International Test Conference, pp. 1159-1168, 2002

42. N. K. jha and S. Gupta, "Testing of Digital Systems", Cambridge University Press, 2003

43. W. X. Li, "Introduction to system programming", School of Computer Science, Carleton University, pp. 57-59, January 2000

44. S. R. Das, M. H. Assaf, and A. Nayak, "Selecting outputs for merger in space compression using concepts of hamming distance and sequence weights", *IASTED International Conference on Modelling, Simulation and Optimization*, Gold Coast, Australia, May 6-9, 1996

45. H. K. Lee and D. S. Ha, "New techniques for improving parallel fault simulation in synchronous sequential circuits", Proc. International Conference on Computer-Aided Design, pp. 10-17, October 1993

46. Y. Zorian, and E. J. Marinissen, "System Chip Test: How Will It Impact Your Design?", pp. 136-141, *Proceedings of the 37th Conference on Design Automation*, Los Angeles, CA, June 5-9, 2000

47. V. D. Agrawal, C.R. Kime and K. K. Saluja, "A tutorial on built-in self-test (part 2)", *IEEE Design and test of Computers*, vol. 10, pp. 69-77, June 1993

48. V. D. Agrawal et al., "Built-in self-test for digital circuits", *AT&T Technical Journal*, vol. 73, pp. 30-39, March/April 1994

49. K. Chakrabarty and J. P. Hayers, "Efficient test response compression for multiple-output circuits", *Proc. 1994 Int. Test Conference*, pp. 501-510, 1994

50. R. G. Daniels and W. B. Bruce, "Built-in self-test trends in Motorola microprocessors", *IEEE Design and Test of Computers*, vol. 2, pp. 64-71, April 1985

51. I. Pomeranz, L. N. Reddy and S. M. Reddy, "Compactest: A method to generate compact test sets for combinational circuits", *Proc. 1991 International Test Conference.* pp. 194-203, 1991

52. D. K. Pradham and S. K. Gupta, "A new framework for designing and analyzing BIST techniques and zero aliasing compression", *IEEE Transactions on Computers*, vol. C-40, pp. 743-763, June 1991

53. S. M. Reddy, K. K. Saluja and M. G. Karpovsky, "A data compression technique for built-in self-test", *IEEE Transactions on Computers*, vol. C-37, pp. 1151-1156, September 1998

54. S. B. Akers, "A parity bit signature for exhaustive testing", *IEEE Trans. Computer Aided Design*, vol. 7, pp. 333-338, March 1988

55. S. R. Das, A. R. Nayak, M. H. Assaf, and W. B. Jone, "Realizing ultimate compression with acceptable fault coverage degradation to reduce MISR size in BIST applications by nonexhaustive test patterns", *1997 International Symp. on Circuit and Systems*, pages 2717-2720, June 1997

56. ISCAS 85 and 89 Benchmark circuits
http://www.fm.vslib.cz/~kes/asic/iscas/

57. IEEE P1500 Web Site.
http://grouper.ieee.org/groups/1500/

58. Inventra core library, Mentor Graphics,
http://www.mentorg.com/inventra/

59. 2004 International Technology Roadmap for Semiconductors,
http://public.itrs.net/

LIST OF PAPERS BY THE CANDIDATE

1. Liwu Jin, "Hardware and software co-design in space compaction of cores-based digital circuits", IMTC 2004 – Instrumentation and Measurement Technology Conference, Como, Italy, 18-20 May 2004, (with Mansor H. Assaf, Sunil R. Das, Emil M. Petriu, Voicu Groza)

2. Liwu Jin, "Testing embeded cores-based system-on-a-chip (SOC) – Test architecture and implementation", MIC 2004, Switzerland, (with Sunil R. Das, Mansor H. Assaf, Emil M. Petriu, Chuan Jin)

3. Liwu Jin, "Implementation of testing environment for digital IP cores", IMTC 2004 – Instrumentation and Measurement Technology Conference, Como, Italy, 18-20 May 2004, (with Sunil R. Das, Chuan Jin, Mansor H. Assaf, Emil M. Petriu, Wen-Ben Jone)

4. Liwu Jin, "Test implementation of embeded cores-based sequencial circuits using Verilog HDL under Altera MAXPLUS II development environment", Turkey IDPT Conference Paper – 2004, (with Sunil R. Das, Chuan Jin, Mansor H. Assaf, Emil M. Petriu, Mehmet Sahinoglu)

APPENDIX

1. Attached program source code

```
*****************************************************
*                                                   *
*       Welcome to Digital Circuit Verification System          *
*                                                   *
*       Writer : Liwu Jin       Systems Science             *
*                                                   *
*       University of Ottawa    Copyright (C) 2003.12       *
*                                                   *
*****************************************************

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <time.h>
#include <iostream.h>
#define max 500
#define lmax 5000
#define wmax 50000
#define start 0
#define interval 1

FILE *fp;
FILE *fp1;
FILE *fp2;
char *getWord(char *word, int lim);
unsigned long int random_number(unsigned long int seed);
void wait();

int main()
{
        char word[lmax];
        char w1[max] = "\0";
        char w2[max] = "\0";
        char filename[max] = "\0";              // testing verilog file name
        char fname[max] = "\0";                 // full file name
        char aline[lmax] = "\0";                // read a line in a file
        char input[lmax] = "\0";                // input name
        char innum[lmax] = "\0";                // input
        char outnum[lmax] = "\0";               // output
        char output[lmax] = "\0";               // ouput name
        char wire[wmax] = "\0";                 // wire name
        char temp[wmax] = "\0";
        char *wd = NULL;
        char *ptr;                              // point
```

```
        char ff;                                // injection "0" or "1"

        int m, n, b, p, gen, seq, fseq;
//      b is testing cycle;
//      m is input number;
//      n is output number;
//      p is wire number
//      gen is the random number

        int i = 0, j = 0, a = 0;
        int found = 1;
        int flag, count, ot;
        int counter[wmax] = {0};                // counter

        unsigned long int seed;
        int c;
//      c is seed number
//      srand(time(NULL));
//      seed = rand();

//      open verilog file you want to test

        printf( "\nEnter File Name\t:\t");
        gets(filename);

        strcpy(fname, filename);
        strcat(fname, ".v");
        fp = fopen(fname, "r");

        if(fp == NULL)
        {
                printf("Could not open file %s!\n", fname);
                exit( 0 );
        }

        printf("Opening file\t:\t\"%s\" \n", fname);

//      get input, output, wire numbers and arrays
        while((getWord(word, max)) != NULL)
        {

                if (strcmp(word, "Ninputs")==0)
                {
                        m=atoi(getWord(word, max)); //m is input number;
                        printf("Input numbers\t:\t%d\n", m);
                }
                else if (strcmp(word, "Noutputs")==0)
                {
                        n=atoi(getWord(word, max)); //n is output number;
                        printf("Output numbers\t:\t%d\n", n);
                }
```

```
            else if (strcmp(word, "Wire")==0)
            {
                    p=atoi(getWord(word, max)); //p is wire number;
                    printf("Wire numbers\t:\t%d\n", p);
            }
            else if (strcmp(word, "input")==0)
            { // printf("Input : \n");
                    for (i=0;i<m;i++) strcat(input, getWord(word, max));
            }
            else if (strcmp(word, "output")==0)
            { // printf("\nOutput : \n");
                    for (i=0;i<n;i++) strcat(output, getWord(word, max));
            }
            else if (strcmp(word, "wire")==0)
            {
                    for (i=0;i<p;i++) strcat(wire, getWord(word, max));
            }

    }

    fclose(fp); // close verilog file

    enter seed number and define test pattern numbers
    printf ("\nEnter seed number\t:\t");
    cin >> c;
    seed = 1050420308;

//  b is the test pattern number
    b = (m+n+p) * 2;
//  printf ("\nEnter test pattern number\t:\t");
//  cin >> b;
    if (b > 199) b = 199;

//  pick up the inputs and outputs

    strcpy(fname, filename);
    strcat(fname, ".vec");
    remove(fname);
    fp = fopen(fname, "a");

    fprintf (fp, "INTERVAL %d ;\n", interval);
    fprintf (fp,"START %d ;\nSTOP %dns ;", start, (b*interval+start));
    fprintf(fp, "\nINPUTS ", i);

    strcpy(temp, input);
    wd = strtok(temp, "NG");

    while (wd != NULL)
    {

//  define "N" or "G" in input, output and wire
```

```
            if (input[0] == 'N')
                    fprintf (fp,"N%s ", wd);
            else
                    fprintf (fp,"G%s ", wd);
            wd = strtok(NULL, "NG");
    }

//      generate random numbers in vector file

    fprintf(fp, ";\nPATTERN\n");

    for (i=0; i<b; i++)
    {

            fprintf(fp, "%d>", i);

            for (int count = 0; count < m; count++)
            {
                    fprintf(fp, " ");
                    gen = random_number(seed) % 2;

                    if (gen == 0)
                            innum[count] = '0';
                    else
                            innum[count] = '1';

                    fprintf(fp, "%c", innum[count]);
                    innum[count] = '\0';
            }

            fprintf(fp, "\n");

    }

    fprintf(fp, ";");

//      find outputs
    a=0;
    strcpy(temp, output);

    wd = strtok(temp, "NG");
    while (wd != NULL)
    {
    printf("%d\tN%s\n", ++a, wd);
            if (input[0] == 'N')
                    fprintf (fp,"\nOUTPUTS N%s", wd);
            else
                    fprintf (fp,"\nOUTPUTS G%s", wd);

            fprintf (fp, " ;\n");
            wd = strtok(NULL, "NG");
```

```
        }

        fclose(fp); // close vector file

//      setup functional snf extractor
        strcpy(fname, filename);
        strcat(fname, ".acf");
        fp = fopen(fname, "w");
        fprintf (fp,"COMPILER_PROCESSING_CONFIGURATION\n
        BEGIN\nFUNCTIONAL_SNF_EXTRACTOR = ON;\nEND;");
        fclose(fp);// close acf file ----------


//      Do complier and simulation
        strcpy(word, "maxplus2 -c -s -vec ");
        strcat(word, filename);
        strcat(word, " -s -vec ");
        strcat(word, filename);
        strcat(word, ".vec -tbl ");
        strcat(word, filename);
        strcat(word, ".tbl ");
        strcat(word, filename);
        system(word);


//      copy table file to backup file
        strcpy(word, "copy ");
        strcat(word, filename);
        strcat(word, ".tbl ");
        strcat(word, filename);
        strcat(word, ".bak");
        system(word);

        time_t t1,t2;
        int difftime = 0;
        (void)time(&t1);


//      FAULT INJECTION IN INPUTS
        printf("\n--- FAULT INJECTION IN INPUTS---\n");

//      pick up data from backup file
        strcpy(fname, filename);
        strcat(fname, ".bak");
        fp = fopen(fname, "r");

        flag = -1;
        seq = 0;
        j = 0;

        for (count=0;count<m;count++)
```

```
{

if (flag<0)
        fj = '0';
else
        fj = '1';

while((fgets(aline, lmax, fp)) != NULL)
{

        if ((ptr = strstr(aline, ">")) != NULL)
        {

                strcpy(aline, ptr+2);

                for (i=0;i<2*m;i++)
                        innum[i] = aline[i];

                if ((ptr = strstr(aline, "=")) != NULL)
                        strcpy(outnum, ptr+2);

//              MAXPLUS2 run on the background

                strcpy(fname, filename);
                strcat(fname, ".vec");
                remove(fname);
                fp1 = fopen(fname, "a");

                fprintf (fp1, "INTERVAL %d ;\n", interval);
                fprintf (fp1,"START %d ;\nSTOP %dns ;", start, 1);
                fprintf(fp1, "\nINPUTS ", i);
                strcpy(temp, input);

                wd = strtok(temp, "NG");
                while (wd != NULL)
                {
//              define "N" or "G" in input, output and wire
                        if (input[0] == 'N')
                                fprintf (fp1,"N%s ", wd);
                        else
                                fprintf (fp1,"G%s ", wd);
                        wd = strtok(NULL, "NG");
                }

//              generate random numbers in vector file
                fprintf(fp1, ";\nPATTERN\n");
                strcpy(temp, innum);

                int i;

                if (temp[2*count] == fj)
```

```
                {
                        ot = 0;
                        goto samefj;
                }
                else
//                      inject fault in specific position
                        temp[2*count] = fj;
                        fprintf(fp1, "0>");

                        for (i = 0; i < 2*m; i++)
                        {
                                fprintf(fp1, "%c", temp[i]);
                                temp[i] = '\0';
                        }

                        fprintf(fp1, "\n");
                        fprintf(fp1, ";");

//              find outputs
                a=0;
                strcpy(temp, output);
                wd = strtok(temp, "NG");

                while (wd != NULL)
                {
                        if (input[0] == 'N')
                                fprintf (fp1,"\nOUTPUTS N%s", wd);
                        else
                                fprintf (fp1,"\nOUTPUTS G%s", wd);

                        fprintf (fp1, " ;\n");
                        wd = strtok(NULL, "NG");

                }

                fclose(fp1); // close vector file

//              setup functional snf extractor
                strcpy(fname, filename);
                strcat(fname, ".acf");
                fp1 = fopen(fname, "w");
                fprintf (fp1,"COMPILER_PROCESSING_CONFIGURATION\n
                BEGIN\nFUNCTIONAL_SNF_EXTRACTOR = ON;\nEND;");

                fclose(fp1);                    // close acf file

//              Do complier and simulation
                strcpy(word, "maxplus2 -s -vec ");
//              strcat(word, filename);
//              strcat(word, " -ta_setup -s -vec ");
                strcat(word, filename);
```

```
strcat(word, ".vec -tbl ");
strcat(word, filename);
strcat(word, ".tbl ");
strcat(word, filename);
system(word);
//          complete the simulation and get table file
//          printf("\n%s", outnum);

strcpy(fname, filename);
strcat(fname, ".tbl");
fp1 = fopen(fname, "r");

while((fgets(temp, lmax, fp1)) != NULL)
{
        if ((ptr = strstr(temp, "0.0>")) != NULL)
        {
                strcpy(temp, ptr+7+2*m);
                break;  //read the output(just one line) in table file then break
        }
}                                       // end while

fclose(fp1);// close temp table file

ot = strcmp(outnum, temp);

samefj:
if (seq == b)
{
        printf("--- THE END OF TEST PATTERN ---");

        break;                          // if reach the end, break
}
else if (ot == 0)
{
        printf("\n*** NO fault found ***");
        fclose(fp1);
        system("del *.vec");
}
else
{
        printf("\n*** ^0^ *** FAULT FOUND *** ^0^ ***\n");
        printf("\n%d:%s\n", flag, outnum);
        printf("\n%d:%s\n", flag, temp);

        j = counter[seq];
        counter[seq] = ++j;
        break;  //
}

seq++; // test pattern number
```

```
        }                                   // end if ((ptr = strstr(aline, ">")) != NULL)

}                                           // end while ((fgets(aline, lmax, fp)) != NULL)


        if (flag<0) count--;

        flag=flag*(-1);
        rewind(fp);
        seq = 0;                            // test pattern number recounts

} // end big for

        fclose(fp);                         // close backup file

        a = 0;
        for (i=0;i<b;i++)
        {
                printf("%d-", counter[i]);
                a = a + counter[i];
        }

        printf("\n--- END OF FAULT INJECTION IN INPUTS ---\n");
//      printf("\n--- TOTAL %d FAULTS FOUND ---", a);
//      end of fault injection in inputs


//      FAULT INJECTION IN OUTPUTS

        strcpy(fname, filename);
        strcat(fname, ".bak");
        fp = fopen(fname, "r");

        flag = -1; // Inject zero first
        seq = 0; // The position of fault found
        j = 0; // add fault number which have been found

        for (count=0;count<n;count++)
        {
                if (flag < 0) ff = '0';
                else ff = '1';

                while((fgets(aline, lmax, fp)) != NULL)
                {

                        if ((ptr = strstr(aline, "=")) != NULL)
                        {
                                strcpy(outnum, ptr+2);

                                if (seq == b)
                                {
```

```
                                printf("--- THE END OF TEST PATTERN ---");

                                break;  //if reach the end, break
                        }

                        else if (outnum[2*count] == fj)
                                printf("*** NO fault found ***");
                        else
                        {
                                j = counter[seq];
                                counter[seq] = ++j;

                                break;  //if found, break while cycle
                        }

                seq++;

                } // end if=

        } // end while line


        if (flag<0) count--;

        flag = flag*(-1);
        rewind(fp);
        seq = 0;

} // end big for

fclose(fp); // close backup file
//      END OF FAULT INJECTION IN OUTPUTS


//      FAULT INJECTION IN WIRES
//      printf("\n--- FAULT INJECTION IN WIRES---\n");


//      pick up data from backup file
        strcpy(fname, filename);
        strcat(fname, ".bak");   // table file
        fp = fopen(fname, "r");

        flag = -1;
        seq = 0;
        fseq = 0;
//      j = 0;

        for (count=0;count<p;count++)
        {
```

```
                if (flag<0)
                        ff = '0';
                else
                        ff = '1';


//              generate vector file
                j = 0;
//              strcpy(fname, filename);
//              strcat(fname, ".bak");              // table file
//              fp = fopen(fname, "r");

                strcpy(fname, filename);
                strcat(fname, ".vec");
                remove(fname);
                fp1 = fopen(fname, "w");

                fprintf (fp1, "INTERVAL %d ;\n", interval);
                fprintf (fp1,"START %d ;\nSTOP %dns ;", start, b);
                fprintf(fp1, "\nINPUTS ", i);
                strcpy(temp, input);

                wd = strtok(temp, "NG");
                while (wd != NULL)
                {
//              define "N" or "G" in input, output and wire
                        if (input[0] == 'N')
                                fprintf (fp1,"N%s ", wd);
                        else
                                fprintf (fp1,"G%s ", wd);
                        wd = strtok(NULL, "NG");
                }

                fprintf(fp1, ";\nPATTERN\n");


        while((fgets(aline, lmax, fp)) != NULL)
        {

                if ((ptr = strstr(aline, ">")) != NULL)
                {

                        strcpy(aline, ptr+2);

                        for (i=0;i<2*m;i++)
                                innum[i] = aline[i];

                        if ((ptr = strstr(aline, "=")) != NULL)
                                strcpy(outnum, ptr+2);
```

```
//              generate random numbers in vector file

                fprintf(fp1, "%d> ", j);
                        for (int i = 0; i < 2*m; i++)
                        {
                                fprintf(fp1, "%c", innum[i]);
                                innum[i] = '\0';
                        }

                fprintf(fp1, "\n");
                j++;
                if (j==b) {
                        fprintf(fp1, ";");
                        break;
                }

        }                                       // end if ((ptr = strstr(aline, ">")) != NULL)
}                                               // end while ((fgets(aline, lmax, fp)) != NULL)


//              find outputs
                a=0;
                strcpy(temp, output);
                wd = strtok(temp, "N");

                while (wd != NULL)
                {
                        fprintf (fp1,"\nOUTPUTS N%s", wd);
                        fprintf (fp1, " ;\n");
                        wd = strtok(NULL, "N");
                }

                fclose(fp1);                    // close vector file

//              vector file generated!

//              inject faults in internal wires

                if (found == 1)
                {

//              find wires
                a=0;
                strcpy(temp, wire);
                wd = strtok(temp, "N");

                while (wd != NULL)
                {
                        if (a == count)
```

```
                    strcpy(word, wd);
//                  word represents specific wire name which depends on COUNT
                    a++;
                    wd = strtok(NULL, "N");
                }


                strcpy(w1, ",N");
                strcat(w1, word);
                strcat(w1, ",");

                strcpy(w2, ",N");
                strcat(w2, word);
                strcat(w2, ")");

//              printf("%s\t%s\n", w1, w2);

//              change verilog file with fault injection

//              pick up data from verilog file

                strcpy(fname, filename);
                strcat(fname, ".v");            // verilog file
                fp2 = fopen(fname, "r");

                strcpy(fname, filename);
                strcat(fname, "a.v");           // verilog file
                remove(fname);
                fp1 = fopen(fname, "a");

                while (fgets(aline, lmax, fp2) != NULL)
                {

                    strcpy(word, "module ");
                    strcat(word, filename);

//                  In verilog file, find module name and append "a" with it
                    if ((ptr = strstr(aline, word)) != NULL)
                    {
                        strcpy(temp, ptr + strlen(word));
                        *ptr = 0;
                        strcat(aline, word);
                        strcat(aline, "a");
                        strcat(aline, temp);
                    }

//                  find "(" at first, then find specific word
                    else if ((ptr = strstr(aline, "(")) != NULL && (ptr = strstr(aline, w1)) !=
NULL)
                    {
                        strcpy(temp, ptr + strlen(w1));
```

```
                    *ptr = 0;

                    if (flag<0)
                            strcpy(word, ",0,");
                    else
                            strcpy(word, ",1,");

                    strcat(aline, word);
                    strcat(aline, temp);
//                  printf("\n%d:%s", count, aline);
                }

                else if ((ptr = strstr(aline, "(")) != NULL && (ptr = strstr(aline, w2)) !=
NULL)
                {
                    strcpy(temp, ptr + strlen(w2));
                    *ptr = 0;

                    if (flag<0)
                            strcpy(word, ",0)");
                    else
                            strcpy(word, ",1)");

                    strcat(aline, word);
                    strcat(aline, temp);
                }

                fprintf(fp1, "%s", aline);

            }                               // end while


            fclose(fp2);                    // temp file
            fclose(fp1);                    // verilog file
            rewind(fp2);

//          setup functional snf extractor
            strcpy(fname, filename);
            strcat(fname, "a.acf");
            fp1 = fopen(fname, "w");
            fprintf (fp1,"COMPILER_PROCESSING_CONFIGURATION\n
            BEGIN\nFUNCTIONAL_SNF_EXTRACTOR = ON;\nEND;");
            fclose(fp1);
//          close acf file ----------


//          Do complier and simulation

            strcpy(word, "maxplus2 -c ");
//          strcat(word, filename);
            strcat(word, "-s -vec ");
```

```
                strcat(word, filename);
                strcat(word, ".vec -tbl ");
                strcat(word, filename);
                strcat(word, "a.tbl ");
                strcat(word, filename);
                strcat(word, "a");
                system(word);
                system("del *.cnf");

//              complete the simulation and get table file

                rewind(fp);
                strcpy(fname, filename);
                strcat(fname, "a.tbl");
                fp1 = fopen(fname, "r");

                while((fgets(temp, lmax, fp)) != NULL)
                {

                        if ((ptr = strstr(temp, ">")) != NULL)
                        {
                                strcpy(temp, ptr+2);


                                if (seq == b)
                                {
                                        printf("\nSEQ::: --- THE END OF TEST PATTERN ---
");
                                        found = 1;
                                        seq = 0;

                                        system("del *.hif");
                                        system("del *.acf");
                                        system("del *.mmf");
                                        system("del *.mtb");
                                        system("del *.tao");
                                        system("del *.scf");
                                        system("del *.mtf");
                                        system("del *.ndb");
                                        system("del *.cnf");
                                        system("del *.pch");
                                        system("del *.pdb");
                                        system("del *.ilk");
                                        system("del *.obj");
                                        system("del *.snf");
                                        system("del *.dls");
                                        system("del *.vec");
                                        system("del *.tbl");
                                        system("del *.bak");
                                        system("del vc60.*");
```

```
        break;  // if reach the end, break
}

else
{
        while((fgets(aline, lmax, fp1)) != NULL)
        {
                if ((ptr = strstr(aline, ">")) != NULL)
                {
                        strcpy(aline, ptr+2);

                        if (fseq == b)
                        {

                                found = 1;
                                fseq = 0;

                                break;  //
        }

                        else if (fseq == seq)
                        {
                                if (strcmp(temp, aline) != 0)
                                {


                                        j = counter[seq];
                                        counter[seq] = ++j;

                                        found = 1;
                                        fseq = 0;
                                        seq = 0;

                                        fclose(fp1);
                                        strcpy(word, "del ");
                                        strcat(word, filename);
                                        strcat(word, "a.*");
                                        system(word);

                                        break;
                                }

                                else
                                {
                                        found = 0;
                                        fseq++;
                                        break;
                                }

                        } // end else if
```

```
                                        } // end if

                                    }       //end while

                                }           // end else

                                if (found == 1)
                                {
//                                      printf("seq:%d:::temp:%s", seq, temp);
//                                      if (flag<0) count--;

//                                      flag=flag*(-1);
                                        rewind(fp);
                                        seq = 0; // test pattern number recounts

                                        break;

                                }

                                seq++;          // test pattern number

                        } // end if

                }       // end while

//              fclose(fp1);
//              close temp table file


                if (flag<0) count--;

                flag=flag*(-1);
                rewind(fp);
                seq = 0;                        // test pattern number recounts


        }                                       // end big for

                fclose(fp1);
                fclose(fp); // close bak file

//              printf("\nSequence:%d", seq);
//              printf("\nInput:%s", innum);
//              printf("\nRight:%s", outnum);
//              printf("Fault:%s\n", temp);


//              end of fault injection in wires
```

```
a = 0;
for (i=0;i<b;i++)
{
//          printf("%d\t", counter[i]);
            a = a + counter[i];
}

double aa, ab = 100.00;
aa = 2*(m+n+p);
ab = (a/aa)*100.00;

printf("\n--- TOTAL %d FAULTS FOUND ---", a);
printf("\n--- END OF FAULT INJECTION ---\n");

(void)time(&t2);
difftime = (int)t2-t1;
printf("TOTAL TIME:\t%d SECONDS", difftime);

//          generate a log file to record data
strcpy(fname, filename);
strcat(fname, ".log");
fp = fopen(fname, "w");

fprintf (fp,"\t\t%s\n", fname);
fprintf (fp,"* Log file for the circuit %s.bench.\n", filename);
fprintf (fp,"Number of faults detected by each test pattern:\n\n");

strcpy(fname, filename);
strcat(fname, ".bak");
fp1 = fopen(fname, "r");

i = 0;
while((fgets(aline, lmax, fp1)) != NULL)
{

        if (i>=b)
                break;

        if ((ptr = strstr(aline, ">")) != NULL)
        {
                strcpy(aline, ptr+2);
                fprintf (fp, "test %d\t:", ++i);
                fprintf (fp, "%d faults detected\t", counter[i-1]);
                fprintf (fp, "%s", aline);
        }

}

fclose(fp1);
```

```
fprintf (fp,"\nEnd of fault simulation.\n");

fprintf (fp,"\n*************************************************\n");
fprintf (fp,"*\t\t\t\t\t\t\t\t*\n");
fprintf (fp,"*\tWelcome to Digital Circuit Verification System\t\t*\n");
fprintf (fp,"*\t\t\t\t\t\t\t\t*\n");
fprintf (fp,"*\tWriter : Liwu Jin\tSystems Science\t\t\t*\n");
fprintf (fp,"*\t\t\t\t\t\t\t\t*\n");
fprintf (fp,"*\tUniversity of Ottawa\tCopyright (C) 2003.5\t\t*\n");
fprintf (fp,"*\t\t\t\t\t\t\t\t*\n");
fprintf (fp,"*************************************************\n\n");

fprintf (fp,"*****\tSUMMARY OF FAULT SIMULATION RESULTS\t****\n\n");

fprintf (fp,"1. \tCircuit Structure\n");
fprintf (fp,"\tName of the circuit\t\t\t:\t%s\n", filename);
fprintf (fp,"\tNumber of gates\t\t\t\t:\t%d\n", m+n+p);
fprintf (fp,"\tNumber of primary inputs\t\t:\t%d\n", m);
fprintf (fp,"\tNumber of primary outputs\t\t:\t%d\n", n);
fprintf (fp,"\tDepth of the circuit\t\t\t:\t%d\n\n", p);

fprintf (fp,"2. \tSimulation parameters\n");
fprintf (fp,"\tSimulation mode\t\t\t:\trandom\n");
fprintf (fp,"\tInitial random number generator seed\t:\t%d\n\n", seed);

fprintf (fp,"3. \tSimulation results\n");
fprintf (fp,"\tNumber of test patterns applied\t\t:\t%d\n", b);
fprintf (fp,"\tFault coverage\t\t\t:\t%7.3f %%\n", ab);
fprintf (fp,"\tNumber of collapsed faults\t\t:\t%d\n", 2*(m+n+p));
fprintf (fp,"\tNumber of detected faults\t\t:\t%d\n", a);
fprintf (fp,"\tNumber of undetected faults\t\t:\t%d\n\n", (2*(m+n+p)-a));

fprintf (fp,"4. \tTotal running time\t\t\t:\t%d seconds\n", difftime);
fprintf (fp,"\tTotal simulation time\t\t\t:\t%d seconds\n", 2*difftime/5);

//      printf ("\nLog file generated!\n");

        fclose(fp);                     // close log file



//      Creat vector file for deterministic testing

//      copy vector file to temp file
        strcpy(word, "copy ");
        strcat(word, filename);
        strcat(word, ".vec ");
        strcat(word, filename);
        strcat(word, ".tmp");
        system(word);
```

```
        strcpy(fname, filename);
        strcat(fname, ".tmp");
        fp = fopen(fname, "r");

        strcpy(fname, filename);
        strcat(fname, ".vec");
        fp1 = fopen(fname, "w");

        i = 0;
        j = 0;
        while (fgets(aline, lmax, fp) != NULL)
        {

                strcpy(word, ">");

//              fprintf(fp, "%d>", i);
//              counter[i] != 0 &&

                if ((ptr = strstr(aline, word)) != NULL)
                {
                        if (counter[i] != 0)
                        {
                                fprintf(fp1, "%d>", j);
                                strcpy(temp, ptr + strlen(word));

                                fprintf(fp1, "%s", temp);
                                j++;
                        }

                        i++;
                }

                else fprintf(fp1, "%s", aline);


        }                                       // end while


        fclose(fp1);
        fclose(fp);

        strcpy(word, "type ");
        strcat(word, filename);
        strcat(word, ".vec");
//      system(word);
```

```
//              Finally, delete all temp files
                printf("\n");
                system("del *.hif");
                system("del *.acf");
                system("del *.mmf");
                system("del *.mtb");
                system("del *.tao");
                system("del *.scf");
                system("del *.mtf");
                system("del *.ndb");
                system("del *.cnf");
                system("del *.pch");
                system("del *.pdb");
                system("del *.ilk");
                system("del *.obj");
                system("del *.snf");
                system("del *.dls");
                system("del *.vec");
                system("del *.tbl");
                system("del *.bak");
                system("del *.tmp");
                system("del vc60.*");

                strcpy(word, "del ");
                strcat(word, filename);
                strcat(word, "a.*");
                system(word);

                strcpy(word, "type ");
                strcat(word, filename);
                strcat(word, ".log");
                system(word);

                return 0;


}

char *getWord(char *word, int lim)
{

        char c=0;
        char *w = word;
        // pass by non-alnum

        while(!isalnum(c = getc(fp)))
                if (c==EOF) return NULL;
        *w++ = c;
        //first character of a word
        while(isalnum(c=getc(fp)) && w<word+lim-1 || c=='_')
```

```
                *w++ = c;
        *w = 0;
        //terminate string
        return word;

}

unsigned long int random_number(unsigned long int seed)
{
        const unsigned long int modulo = 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) % modulo;
        return ran;

}
```

2. Samples of testing files: vector file and log file

C432 vector file sample:

INTERVAL 1 ;
START 0 ;
STOP 199ns ;
INPUTS N1 N4 N8 N11 N14 N17 N21 N24 N27 N30 N34 N37 N40 N43 N47 N50 N53 N56 N60 N63 N66 N69 N73 N76 N79 N82 N86 N89 N92 N95 N99 N102 N105 N108 N112 N115 ;
PATTERN
0> 0 1 1 1 0 0 0 0 1 0 0 1 1 1 0 0 0 0 0 1 0 1 0 1 0 1 0 1 1 0 0 0 0 0
1> 0 0 1 0 1 0 1 0 1 1 0 0 0 0 1 1 1 1 0 1 1 1 0 1 0 1 1 0 1 0 1 0 1 1 0 1
2> 0 1 0 1 1 1 1 1 1 1 1 0 1 1 0 0 0 0 0 1 0 0 0 0 0 1 1 0 1 1 0 0 1 0 0 0 0
3> 1 1 0 0 0 1 1 0 1 1 1 0 0 0 0 0 1 1 0 0 1 1 0 0 1 0 0 0 1 1 1 0 0 0 0 0
4> 1 0 0 1 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0 1 0 1 0 0
5> 1 0 0 0 0 0 0 1 1 0 0 1 0 1 1 1 1 0 1 0 0 1 1 0 0 1 1 0 1 1 0 1 0 0 0 1
6> 0 0 0 0 0 1 0 1 0 1 0 0 1 0 1 1 1 0 0 0 0 1 0 1 0 0 0 1 0 1 1 0 1 0 0 0
7> 0 1 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 1 0 1 1 1 0 0 0 0
8> 0 1 1 0 0 1 1 0 1 1 1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 0 1 0 0 1 0 1 0 0 1 1 1
9> 1 1 1 0 0 1 0 0 1 1 0 1 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0 1 0 0 1 1 1 0 0
10> 0 0 1 1 0 1 1 1 0 0 0 1 1 0 0 1 1 0 0 1 0 1 0 0 1 1 0 1 0 1 1 0 1 0 1 0
11> 0 0 0 0 0 0 1 1 0 0 1 0 1 0 1 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 1 1 0 1 1 0
12> 1 1 1 1 1 1 1 0 0 1 1 0 1 1 0 0 1 1 1 0 0 1 0 1 1 1 0 0 1 0 0 1 0 1 0 0
13> 1 0 1 1 0 0 1 1 0 0 1 0 0 0 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 0 0 1 0 1 0 1
14> 0 0 0 0 1 0 1 1 0 0 1 1 1 0 0 0 1 1 1 1 1 0 0 0 1 0 1 0 1 1 0 1 1 1 1 1
15> 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0 1 1 1 1 1 0 0 1 0 0 0 1 0 0 0 0 1 1 0 0 1
16> 1 1 1 1 0 1 0 0 1 0 0 1 1 0 1 0 0 0 0 1 0 1 1 0 1 0 1 1 0 1 1 0 0 0 1 1
17> 1 0 0 1 0 1 0 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 0 1 0 0 0 1 1 0 0 0 1 0
18> 0 0 1 1 0 1 1 1 1 0 1 0 1 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 0 1 1 1 1 0 1 1 0 1
19> 1 1 0 1 0 1 1 0 1 0 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 1 1 0 0 1 0 1 0 1 0 0
20> 1 0 1 0 1 1 0 1 0 1 0 0 1 0 0 1 0 1 0 1 0 0 1 0 1 1 1 1 1 1 0 1 1 1 1 0 1
21> 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 0 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1 1 0 0 0 0
22> 0 0 0 0 1 0 0 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0 0 0 0 1 1 1 1
23> 0 1 1 1 0 1 1 0 0 0 1 1 1 0 1 1 0 0 0 0 1 1 0 0 0 0 0 1 1 1 1 0 0 1 1 1
24> 1 0 1 1 1 0 0 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 0 1 1 0 1 1
25> 1 0 0 1 1 0 1 1 0 1 0 1 1 1 1 1 1 1 0 1 0 1 0 0 0 0 0 1 0 0 0 1 1 0 0 1 1
26> 1 1 1 1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 0 1 1 0 0 0 1 1 1 1 1 1 0 1 1 0 1 0 1
27> 0 0 1 1 1 1 0 0 1 0 0 1 1 0 1 1 0 0 0 0 1 0 1 1 1 1 1 1 1 1 0 1 0 0 1 1 0
28> 1 0 0 1 0 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 1 1 0 1 1 0 1 0 0 1 0 0 1
29> 0 1 0 1 0 0 0 1 1 0 0 0 0 0 1 1 0 0 0 0 1 0 1 1 1 0 0 0 0 1 1 0 0 1 0 1
30> 0 1 1 1 0 1 1 0 0 1 1 0 1 0 1 1 1 1 1 0 1 0 0 1 0 1 1 1 1 1 1 1 0 1 1 0 0 0 1 1
31> 0 0 1 1 1 0 0 0 0 1 0 0 1 1 0 1 0 1 1 0 0 1 0 0 1 1 1 1 1 1 1 0 0 0 1 1 0
32> 0 0 1 1 0 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0 1 1 1 0 1 0 0
33> 0 1 0 0 0 0 1 1 0 0 1 1 1 1 0 0 0 1 0 1 0 1 1 1 0 0 1 1 1 0 1 0 0 1 0 1
34> 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 1 1 0 1 1 1 1 0 0 0 0 0 0 0 0 1 0 0 1 0
35> 0 1 0 1 0 0 0 0 0 0 1 0 1 1 1 1 0 1 1 1 1 0 0 1 0 1 0 0 1 0 0 1 1 0 0 0
36> 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1
37> 1 0 1 1 1 0 1 0 1 0 0 0 1 1 1 1 0 0 0 0 0 1 0 0 1 0 1 1 0 1 1 0 0 1 0 0 0 0
38> 0 0 0 1 0 1 0 1 1 0 0 1 0 1 1 1 0 1 0 0 1 0 0 1 0 1 0 1 1 1 0 1 0 1 1 1
39> 1 1 0 1 0 0 1 1 0 1 0 0 0 1 1 0 1 1 0 1 0 1 0 0 1 0 0 1 1 1 1 1 1 0 1 1 1

40> 1 0 1 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 0 1 1 0 0 0 0 1
41> 0 0 1 1 0 0 1 0 1 0 0 1 1 0 1 1 1 1 1 1 1 0 0 0 0 0 0 1 1 0 1 0 0 0 1 1
42> 1 0 1 1 0 1 1 1 1 1 1 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0
43> 0 1 0 1 0 0 0 1 1 0 1 1 1 0 1 1 1 0 0 0 1 1 0 1 1 1 1 1 1 0 0 0 0 0 1 0 0
44> 1 0 0 0 0 1 1 0 0 1 0 1 1 1 0 1 0 0 0 1 0 0 1 1 0 1 0 0 0 0 1 0 1 1 0 1
45> 0 1 1 0 0 0 1 1 0 0 0 1 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 1 1 1 0 0 0 1 1 0
;
OUTPUTS N223 ;

OUTPUTS N329 ;

OUTPUTS N370 ;

OUTPUTS N421 ;

OUTPUTS N430 ;

OUTPUTS N431 ;

OUTPUTS N432 ;

C432 log file sample:

C432.log
** Log file for the circuit C432.bench.*
Number of faults detected by each test pattern:

test 1 :70 faults detected *0 1 1 1 0 0 0 0 1 0 0 1 1 1 0 0 0 0 0 1 0 1 0 1 0 1 0 1 1 0 0 0 0*
0 = 1 0 1 0 0 0 0
test 2 :25 faults detected *0 0 1 0 1 0 1 0 1 1 0 0 0 0 1 1 1 1 0 1 1 1 0 1 0 1 1 0 1 0 1 0 1 1 0*
1 = 1 1 1 1 1 1 0
test 3 :37 faults detected *0 1 0 1 1 1 1 1 1 1 0 1 1 0 0 0 0 0 1 0 0 0 0 0 1 1 0 1 1 0 0 1 0 0 0*
0 = 1 1 0 0 0 1 0
test 4 :32 faults detected *1 1 0 0 0 1 1 0 1 1 1 0 0 0 0 0 1 1 0 0 1 1 0 0 1 0 0 0 1 1 1 0 0 0 0*
0 = 1 1 0 1 1 0 0
test 5 :20 faults detected *1 0 0 1 0 0 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0 1 0 1 0*
0 = 1 1 0 1 0 1 1
test 6 :23 faults detected *1 0 0 0 0 0 0 1 1 0 0 1 0 1 1 1 1 0 1 0 0 1 1 0 0 1 1 0 1 1 0 1 0 0 0*
1 = 1 1 1 1 0 0 1
test 7 :9 faults detected *0 0 0 0 0 1 0 1 0 1 0 0 1 0 1 1 1 0 0 0 0 1 0 1 0 0 0 1 0 1 1 0 1 0 0*
0 = 1 1 1 1 1 1 1
test 8 :0 faults detected *0 0 1 0 0 0 1 0 1 1 0 0 0 0 1 1 1 0 0 1 0 0 0 0 0 1 1 0 1 1 1 0 0 1 0*
1 = 1 1 1 1 1 1 0
test 9 :2 faults detected *0 1 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 1 0 1 1 1 0 0 0*
0 = 1 1 1 1 1 1 0
test 10 :8 faults detected *0 1 1 0 0 1 1 0 1 1 1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 0 1 0 0 1 0 1 0 0 1 1*
1 = 1 1 0 1 0 1 0
test 11 :2 faults detected *1 1 1 0 0 1 0 0 1 1 0 1 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0 1 0 0 1 1 1 0*
0 = 1 1 1 1 1 1 0
test 12 :6 faults detected *0 0 1 1 0 1 1 1 0 0 0 1 1 0 0 1 1 0 0 1 0 1 0 0 1 1 0 1 0 1 1 0 1 0 1*
0 = 1 1 1 1 0 1 0
test 13 :4 faults detected *0 0 0 0 0 0 1 1 0 0 1 0 1 0 1 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 1 1 0 1 1*
0 = 1 0 1 1 0 1 0
test 14 :17 faults detected *1 1 1 1 1 1 1 0 0 1 1 0 1 1 0 0 1 1 1 0 0 1 0 1 1 1 0 0 1 0 0 1 0 1 0*
0 = 1 1 0 1 1 0 1
test 15 :10 faults detected *1 0 1 1 0 0 1 1 0 0 1 0 0 0 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 0 0 1 0 1 0*
1 = 1 0 0 1 0 1 0
test 16 :3 faults detected *0 0 0 0 1 0 1 1 0 0 1 1 1 0 0 0 1 1 1 1 1 0 0 0 1 0 1 0 1 1 0 1 1 1 1*
1 = 1 1 0 1 0 0 1
test 17 :6 faults detected *0 0 0 0 1 0 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 0 0 1 0 0 0 1 0 0 0 0 1 1 0 0*
1 = 1 1 0 1 1 0 1
test 18 :17 faults detected *1 1 1 1 0 1 0 0 1 0 0 1 1 0 1 0 0 0 0 1 0 1 1 0 1 0 1 1 0 1 1 0 0 0 1*
1 = 0 1 0 1 1 1 1
test 19 :3 faults detected *1 0 0 1 0 1 0 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 0 1 0 0 0 1 1 0 0 0 1*
0 = 1 0 1 1 0 1 1
test 20 :0 faults detected *1 1 0 1 0 1 1 1 0 0 1 0 0 0 1 0 0 0 0 0 1 1 0 1 1 1 1 1 0 0 1 1 0 1 0*
0 = 1 1 0 1 0 1 1
test 21 :0 faults detected *0 1 1 0 0 1 1 0 1 0 1 1 0 1 1 1 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0*
1 = 1 1 1 1 0 0 1

test 22 :15 faults detected	*0 0 1 1 0 1 1 1 0 1 0 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 0 1 1 1 0 1 1 0*
1 = 1 1 0 1 0 0 0	
test 23 :0 faults detected	*0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 1 0 0 1 0 0 1 0 1 0 1 1 1 0 0 1 1 0 1 0*
1 = 1 1 1 1 1 1 1	
test 24 :1 faults detected	*1 1 0 1 0 1 1 0 1 0 1 0 0 0 1 1 0 0 0 0 0 0 1 0 1 1 1 0 0 1 0 1 0 1 0*
0 = 1 1 1 1 0 0 1	
test 25 :0 faults detected	*0 0 0 1 0 0 0 1 1 0 1 0 0 1 0 1 1 1 0 0 1 0 1 0 0 0 0 1 0 0 0 1 0 0 0*
0 = 1 1 0 1 1 0 1	
test 26 :7 faults detected	*1 0 1 0 1 1 0 1 0 1 0 0 1 0 0 1 0 1 0 1 0 0 1 0 1 1 1 1 1 0 1 1 1 1 0*
1 = 1 1 1 1 1 1 1	
test 27 :1 faults detected	*0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 0 1 1 0 1 1 0 1 1 0 1 1 1 1 0 1 1 0 0 0*
0 = 1 1 1 1 1 1 0	
test 28 :0 faults detected	*0 1 1 1 1 1 0 1 0 1 0 0 1 0 0 1 1 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0 1 0 0*
1 = 1 0 0 0 0 0 0	
test 29 :4 faults detected	*0 0 0 0 1 0 0 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0 0 0 0 1 1 1*
1 = 1 0 1 1 0 1 0	
test 30 :0 faults detected	*1 1 1 1 1 0 1 1 1 1 1 0 0 0 0 0 1 0 1 1 0 1 1 0 1 0 0 1 1 0 1 0 1 1 1*
1 = 1 0 0 1 0 0 0	
test 31 :5 faults detected	*0 1 1 1 0 1 1 0 0 0 1 1 1 0 1 1 0 0 0 0 1 1 0 0 0 0 0 1 1 1 1 0 0 1 1*
1 = 1 1 1 1 0 1 1	
test 32 :0 faults detected	*1 0 1 1 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 0 1 0 1*
1 = 1 1 0 1 0 1 0	
test 33 :0 faults detected	*0 0 0 0 0 0 0 1 0 1 1 0 1 0 1 1 0 1 1 0 1 0 0 0 0 1 1 1 0 0 1 0 0 0 0*
1 = 1 0 1 1 0 1 0	
test 34 :0 faults detected	*1 1 1 1 0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 0 0 0 1 0 1 0 0 0 0 0 0 1 0 1 1 0*
1 = 1 1 0 1 0 0 0	
test 35 :3 faults detected	*1 0 1 1 1 0 0 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 0 1 1 0 1*
1 = 1 0 0 1 1 0 0	
test 36 :0 faults detected	*0 1 1 0 1 1 0 0 1 1 1 0 1 0 0 1 1 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 1 1*
1 = 1 1 1 1 0 1 1	
test 37 :0 faults detected	*0 1 1 1 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 1 0 1 1 1 1 0 1 0 1 0 0 1*
1 = 1 0 1 1 0 1 0	
test 38 :0 faults detected	*0 0 1 1 1 0 1 0 0 1 1 1 0 0 1 0 1 0 1 0 0 1 0 0 0 1 1 0 1 0 0 0 1 1 0*
0 = 1 1 1 1 0 1 1	
test 39 :0 faults detected	*0 1 0 0 1 1 0 1 0 1 1 0 0 0 0 0 0 1 1 0 0 0 1 0 1 0 0 0 1 0 1 1 1 1 0*
1 = 1 1 1 1 1 1 1	
test 40 :0 faults detected	*1 0 1 1 1 1 0 0 0 0 1 0 0 0 1 1 0 1 1 1 0 1 1 0 1 0 0 1 1 1 1 0 1 0 0*
1 = 0 1 1 1 1 1 1	
test 41 :3 faults detected	*1 0 0 1 1 0 1 1 0 1 0 1 1 1 1 1 1 0 1 0 1 0 0 0 0 0 1 0 0 0 1 1 0 0 1*
1 = 0 1 0 1 1 1 0	
test 42 :0 faults detected	*1 1 0 0 1 1 0 0 0 0 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 1 1 1 0 1 1 1 1 0 1*
0 = 1 1 1 1 1 1 1	
test 43 :0 faults detected	*1 0 1 0 1 0 0 0 1 1 0 0 0 0 1 0 1 1 0 1 0 1 1 1 1 1 0 0 1 0 0 0 0 1 1*
0 = 1 1 1 1 1 1 0	
test 44 :4 faults detected	*1 1 1 1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 0 1 1 0 0 0 1 1 1 1 1 0 1 1 0 1 0*
1 = 1 1 0 1 1 0 0	
test 45 :6 faults detected	*0 0 1 1 1 1 0 0 1 0 0 1 1 0 1 1 0 0 0 0 1 0 1 1 1 1 1 1 1 0 1 0 0 1 1*
0 = 1 0 1 1 0 0 0	
test 46 :1 faults detected	*1 0 0 1 0 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 1 1 0 1 1 0 1 0 0 1 0 0*
1 = 0 1 0 1 0 0 1	

test 47 :0 faults detected	*0 0 0 0 0 1 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1 0 1 0 0 1 1 1 0 1 0 1 1 1 0*
0 = 1 1 1 1 1 1 1	
test 48 :0 faults detected	*1 0 0 1 1 0 0 1 1 0 0 1 0 1 1 1 0 0 1 1 1 1 0 0 1 1 0 0 0 0 1 0 0 1 0*
1 = 1 1 1 1 0 1 0	
test 49 :0 faults detected	*1 0 1 0 0 0 1 1 0 0 0 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0*
0 = 1 1 1 1 0 0 0	
test 50 :0 faults detected	*0 1 1 0 1 1 0 1 1 0 1 1 0 1 0 0 1 1 1 1 1 1 1 0 0 0 0 1 0 1 0 0 0 1 0*
0 = 1 1 1 1 0 0 0	
test 51 :0 faults detected	*0 1 1 0 0 0 1 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 1 0 0 0 1*
0 = 1 0 1 0 0 0 0	
test 52 :0 faults detected	*1 1 1 0 1 1 0 1 1 1 1 0 0 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 1 1 1*
1 = 1 1 0 1 1 1 1	
test 53 :0 faults detected	*0 1 1 0 0 0 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 0 0 1 1 0*
0 = 1 1 1 1 1 1 0	
test 54 :0 faults detected	*0 0 0 0 1 0 1 0 1 1 1 0 0 0 0 1 0 0 0 1 1 0 1 1 1 0 1 1 1 1 1 0 1 1 0*
1 = 1 1 0 1 0 0 0	
test 55 :0 faults detected	*1 0 1 0 0 0 1 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 1 0 1 1 1 0 1 1*
1 = 1 1 1 1 1 1 0	
test 56 :0 faults detected	*0 1 0 1 1 1 1 1 0 0 0 0 0 0 1 1 0 1 0 1 1 1 1 0 1 0 1 1 1 0 1 0 1 1 0*
0 = 1 1 1 1 0 0 0	
test 57 :2 faults detected	*0 1 0 1 0 0 0 1 1 0 0 0 0 0 1 1 0 0 0 0 1 0 1 1 1 0 0 0 0 1 1 0 0 1 0*
1 = 1 1 1 0 0 0 0	
test 58 :4 faults detected	*0 1 1 1 0 1 1 0 0 1 1 0 1 1 1 1 0 1 0 0 1 0 1 1 1 1 1 1 0 1 1 0 0 0 1*
1 = 1 0 1 0 1 0 1	
test 59 :0 faults detected	*0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 1 0 1 0 0 0 1 0 1 0 1 0 0 1 0 0 0 1 1*
1 = 1 1 1 1 0 0 1	
test 60 :0 faults detected	*0 1 1 1 1 0 1 0 0 1 0 1 0 0 1 0 1 0 0 1 1 1 1 0 0 1 0 1 0 0 1 1 1 1 0*
0 = 1 1 1 1 1 1 0	
test 61 :0 faults detected	*0 0 1 0 1 1 0 1 1 1 0 1 0 0 1 0 1 1 1 0 0 0 1 1 1 1 0 1 0 1 1 1 0 0 1*
0 = 1 1 0 1 1 1 1	
test 62 :0 faults detected	*1 0 1 0 0 0 1 0 1 0 1 1 0 1 1 0 1 0 0 0 0 1 0 0 1 0 1 1 1 0 1 0 0 1 0*
1 = 1 1 0 1 0 1 1	
test 63 :5 faults detected	*0 0 1 1 1 0 0 0 0 1 0 0 1 1 0 1 0 1 1 0 0 1 0 0 1 1 1 1 1 1 1 0 0 0 1 1*
0 = 1 1 1 1 1 0 1	
test 64 :0 faults detected	*1 1 1 1 0 1 0 0 1 0 0 1 0 1 1 0 1 1 1 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0 1*
1 = 1 1 1 1 0 1 1	
test 65 :0 faults detected	*1 0 1 0 0 1 1 0 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 1 0*
1 = 1 1 1 1 1 1 0	
test 66 :1 faults detected	*0 0 1 1 0 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0 1 1 1 0 1 0*
0 = 1 0 1 1 0 0 1	
test 67 :1 faults detected	*0 1 0 0 0 0 1 1 0 0 1 1 1 1 0 0 0 1 0 1 0 1 1 1 0 0 1 1 1 0 1 0 0 1 0*
1 = 1 1 1 0 1 0 0	
test 68 :0 faults detected	*1 1 0 1 1 0 1 0 0 1 0 0 1 1 1 0 1 0 0 0 1 0 0 1 1 0 0 0 0 0 1 0 1 1 0*
0 = 1 1 1 1 0 0 0	
test 69 :0 faults detected	*0 0 1 1 1 1 1 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0 1 1 0 0 1 0 0 0 1 1*
1 = 1 1 1 1 0 0 1	
test 70 :1 faults detected	*0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 1 1 0 1 1 1 0 0 0 0 0 0 0 0 1 0 0 1*
0 = 1 0 1 1 0 1 1	
test 71 :0 faults detected	*1 0 0 1 0 0 1 1 0 1 1 1 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 1 1 0 1 1*
1 = 1 1 1 1 0 1 1	

test 72 :1 faults detected
0 = 1 1 1 0 0 0 0
0 1 0 1 0 0 0 0 0 1 0 1 1 1 1 0 1 1 1 1 0 0 1 0 1 0 0 1 0 0 1 1 0 0

test 73 :0 faults detected
0 = 1 1 1 1 1 1 0
1 0 0 1 1 0 1 0 0 1 0 0 0 1 1 0 1 0 0 0 1 1 1 0 1 1 1 0 0 1 0 1 1 1 0

test 74 :0 faults detected
0 = 1 1 1 1 0 1 0
1 1 0 1 0 1 0 1 0 1 1 1 1 0 0 0 1 1 1 0 1 1 1 0 1 1 0 0 0 1 1 0 0 1 1

test 75 :0 faults detected
0 = 1 1 0 1 1 0 1
1 1 0 0 1 0 0 0 0 1 1 0 1 1 0 1 1 0 0 1 1 0 0 1 0 1 1 0 1 0 0 1 1 0 1

test 76 :0 faults detected
1 = 1 0 1 1 0 1 1
0 0 1 0 1 0 0 0 1 1 1 0 1 0 1 1 0 0 1 0 1 1 1 1 0 0 1 0 1 1 1 1 1 1 0

test 77 :0 faults detected
1 = 1 1 0 1 0 0 1
0 0 1 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1 1 0 1 0 1 1 1 0 0 0 0 1 0 1 1 1 1

test 78 :0 faults detected
1 = 1 1 1 1 0 1 1
0 0 0 1 0 0 1 1 1 0 1 0 1 1 0 0 1 1 0 0 1 1 0 0 0 1 0 0 1 1 1 1 1 1 1

test 79 :0 faults detected
0 = 1 1 1 1 0 1 1
1 0 1 0 1 1 1 1 0 1 1 1 0 0 0 0 0 1 1 0 1 1 0 1 0 1 0 1 0 0 0 1 0 0 1

test 80 :3 faults detected
1 = 0 1 0 0 0 0 0
1 1 0 1 1 1 1 1 1 0 1 1 1 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 1

test 81 :0 faults detected
1 = 1 1 1 0 0 0 0
0 1 0 1 0 0 0 0 1 0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0 1 0 1 0 0 0 1 0 1 0

test 82 :0 faults detected
1 = 0 1 0 1 0 0 0
0 0 0 0 0 0 1 1 0 1 1 0 0 0 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 1 1 0 1 0

test 83 :0 faults detected
1 = 1 1 1 1 1 0 1
0 1 1 0 0 1 1 0 1 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 1 1 1 1 1 1 0 0 1 0 0

test 84 :0 faults detected
1 = 1 1 0 1 0 0 1
0 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 0 0 0 1 0 1 1 1 0 1 0 0 1 0 0 1 0 0

test 85 :0 faults detected
1 = 1 1 1 1 0 0 1
1 0 1 1 0 1 1 1 1 1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 1 1 1 0 0 1 0 1 0 0 1

test 86 :0 faults detected
0 = 1 0 1 1 0 0 0
0 0 0 1 1 1 1 1 0 0 1 1 1 1 0 1 1 0 0 1 1 0 1 0 1 0 1 0 0 0 1 0 1 0 0 1 1

test 87 :0 faults detected
0 = 1 1 1 1 0 1 1
1 1 0 0 0 1 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 0 0 1 1 1 1 0 0 0 0 0

test 88 :0 faults detected
1 = 1 1 0 1 0 0 0
1 0 0 0 0 0 0 1 1 0 1 1 1 1 0 0 0 0 1 0 1 0 0 0 0 0 1 1 1 1 1 0 0 1 0

test 89 :0 faults detected
0 = 1 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1 1 1 0 1 0 0 1 1 0 0 0 0 1 1 0 1 1 0 1 1 0 0 1 0 1 1

test 90 :0 faults detected
1 = 1 1 1 1 1 0 1
0 1 0 1 1 1 1 0 0 0 0 0 1 1 0 1 0 0 1 1 1 1 0 1 0 1 1 1 1 0 0 0 0 0 0 0

test 91 :0 faults detected
0 = 1 1 1 1 1 1 1
1 0 1 0 0 1 0 0 0 1 0 0 1 0 0 1 1 0 1 0 1 0 1 1 0 1 1 1 1 0 0 0 0 1 1

test 92 :0 faults detected
1 = 1 0 0 1 0 0 1
1 1 1 1 0 0 0 0 1 0 0 1 0 0 0 1 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1 0 1 0 0

test 93 :0 faults detected
0 = 1 0 1 1 0 1 1
0 0 0 1 1 0 1 1 0 1 0 1 0 1 1 0 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 0 1 1

test 94 :0 faults detected
1 = 1 1 1 1 0 0 1
1 0 0 1 1 1 1 0 0 0 0 1 0 0 0 1 1 0 0 0 1 0 1 1 1 0 0 0 1 1 0 0 0 0 1

test 95 :0 faults detected
0 = 1 1 1 1 0 1 0
0 1 1 0 1 0 0 0 0 1 1 0 1 1 0 0 1 1 1 0 0 0 1 0 1 1 0 1 0 0 1 1 0 1 1

test 96 :0 faults detected
1 = 1 1 1 1 1 0 1
1 0 1 0 0 1 1 1 0 1 0 0 1 1 0 1 0 0 0 0 1 0 1 1 0 1 0 1 0 0 1 1 1 0 1

test 97 :1 faults detected
0 = 1 1 0 1 0 1 1
1 0 1 1 1 0 1 0 1 0 0 0 1 1 1 1 0 0 0 0 0 1 0 0 1 0 1 1 0 1 1 0 0 0 0

test 98 :0 faults detected
1 = 1 1 1 1 0 1 1
1 0 0 0 0 0 1 0 1 0 1 0 1 1 1 0 1 0 0 0 1 1 0 1 0 1 0 0 1 1 0 0 1 0 1

test 99 :0 faults detected
0 = 0 1 1 1 0 1 1
1 0 1 0 1 0 1 0 0 0 1 1 0 0 1 1 1 0 0 1 0 1 0 1 0 0 0 1 1 0 0 0 1 0 1

test 100 :1 faults detected
1 = 0 1 1 1 0 0 1
0 0 0 1 0 1 0 1 1 0 0 1 0 1 1 1 0 1 0 0 1 0 0 1 0 1 0 1 1 1 0 1 0 1 1

test 101 :0 faults detected
1 = 1 1 0 1 0 1 1
1 1 0 1 1 0 0 1 0 1 1 0 0 0 0 1 0 0 0 0 1 1 0 1 1 0 1 0 0 1 1 0 0 1 1

test 102 :0 faults detected
1 = 1 0 1 1 1 1 0
1 1 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 0 1 1 0 0 1 1 1 1 1 0 0 0 1 1 0 0 0

test 103 :0 faults detected
0 = 1 1 1 1 0 0 1
1 0 0 1 0 0 0 1 0 1 0 1 1 1 1 1 1 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 1 1

test 104 :0 faults detected
0 = 1 1 1 1 0 1 0
0 0 1 0 0 0 1 0 1 0 1 1 1 0 0 1 1 1 0 0 1 0 0 0 1 1 0 1 0 0 0 0 0 0 1

test 105 :0 faults detected
0 = 1 1 1 1 1 1 1
0 1 1 0 1 1 0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0 0 1 1 0 0 1 0 0 0 1 0 1 1

test 106 :0 faults detected
0 = 1 1 0 1 1 1 1
1 1 1 0 1 1 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 1 1 1 0 1 0 1 1 1 1 1 1

test 107 :0 faults detected
0 = 1 1 0 1 1 0 0
0 1 1 0 0 0 0 1 0 1 1 0 0 0 1 0 0 1 0 1 1 1 0 1 0 0 1 1 1 0 0 1 0 1 0

test 108 :0 faults detected
1 = 1 1 0 1 1 1 1
1 1 0 0 0 1 0 0 1 1 0 0 1 0 1 0 1 0 0 0 1 0 0 1 1 1 1 0 0 0 0 1 0 1 0

test 109 :0 faults detected
0 = 1 1 0 1 0 1 0
1 0 0 0 1 0 0 1 0 0 1 1 1 1 0 1 1 0 1 0 1 0 0 0 1 1 0 0 1 1 1 1 0 0 1

test 110 :0 faults detected
0 = 1 1 1 1 1 0 0
1 1 0 1 1 1 1 0 0 1 0 1 1 1 1 0 0 1 0 0 0 1 1 1 0 1 0 1 0 0 0 0 1 1 0

test 111 :0 faults detected
0 = 1 1 1 1 1 1 0
1 1 1 1 1 0 1 0 0 1 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1

test 112 :0 faults detected
0 = 1 1 0 0 0 0 0
0 1 0 0 1 0 1 0 1 0 1 0 0 0 0 1 1 1 0 0 1 1 1 1 1 1 1 0 1 1 1 0 0 0 0

test 113 :5 faults detected
1 = 1 1 1 1 1 0 0
1 1 0 1 0 0 1 1 0 1 0 0 0 1 1 0 1 1 0 1 0 1 0 0 1 0 0 1 1 1 1 1 0 1 1

test 114 :0 faults detected
0 = 1 1 1 1 1 1 1
1 0 1 0 1 1 0 1 0 1 0 0 0 0 1 1 0 0 0 1 1 1 1 0 1 0 0 1 1 1 0 0 1 1 0

test 115 :0 faults detected
0 = 1 0 1 0 1 1 0
0 1 1 1 0 1 0 0 1 1 1 1 0 0 1 1 0 0 1 0 1 0 1 1 1 1 0 1 0 0 0 1 1 1 0

test 116 :1 faults detected
1 = 1 1 0 1 1 1 0
1 0 1 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 0 1 1 0 0 0 0

test 117 :0 faults detected
1 = 1 1 1 1 1 1 1
1 0 0 0 0 1 0 1 0 1 0 0 0 0 1 1 0 1 0 0 0 1 0 1 1 1 1 1 1 1 0 0 0 0 0

test 118 :0 faults detected
1 = 1 1 1 1 0 0 1
1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 0 1 0 0 1 0 1 1 0 1 0 0 1

test 119 :0 faults detected
1 = 1 0 1 1 0 0 1
1 1 1 0 1 0 1 0 1 0 1 1 1 0 0 1 0 1 0 1 1 0 0 1 0 1 1 0 0 1 1 0 0 0 0

test 120 :0 faults detected
1 = 1 1 1 1 1 0 1
0 0 0 1 0 1 1 0 1 0 0 0 1 1 0 1 0 1 0 1 1 0 1 0 0 0 0 0 1 1 0 0 1 0 0

test 121 :0 faults detected
0 = 1 1 0 1 1 1 1
1 1 0 0 0 1 0 1 1 1 0 1 1 0 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 1 1 1 0

test 122 :0 faults detected *0 1 1 1 1 0 1 1 1 1 1 0 0 0 0 1 1 0 0 1 1 0 0 0 1 0 1 0 1 1 0 0 0 1 0*
1 = 1 1 1 1 0 0 1
test 123 :0 faults detected *1 0 0 1 1 0 0 1 1 0 0 1 0 0 0 0 1 0 1 0 0 0 1 1 1 1 0 1 1 0 0 0 0 1 1*
1 = 1 0 0 1 0 0 0
test 124 :0 faults detected *1 0 0 1 1 0 0 1 0 0 0 1 0 1 1 1 0 0 0 1 0 1 0 1 0 1 1 0 1 0 1 1 1 0 0*
0 = 0 1 1 1 0 1 1
test 125 :0 faults detected *1 1 0 1 0 0 0 1 1 1 0 1 0 0 1 1 0 1 1 1 0 0 1 1 1 0 0 0 0 0 1 0 0 1 1*
1 = 1 0 0 1 0 0 0
test 126 :0 faults detected *0 1 0 0 0 0 1 1 0 1 0 0 0 1 0 1 0 0 0 0 1 0 1 0 1 0 1 1 0 0 0 0 1 1 0*
0 = 1 1 1 0 1 0 1
test 127 :0 faults detected *1 1 0 1 0 1 1 0 0 1 1 1 0 1 1 1 0 0 1 0 0 0 0 0 0 1 0 1 0 0 1 1 1 1 0*
1 = 1 1 1 1 0 1 0
test 128 :0 faults detected *1 0 1 1 0 0 1 0 0 0 0 1 1 1 1 1 1 1 0 1 0 1 0 0 1 1 0 1 1 1 1 1 0 1 0 0*
1 = 1 1 0 1 0 1 0
test 129 :0 faults detected *1 1 1 0 0 0 0 1 1 0 1 1 1 0 1 0 1 1 1 1 1 0 1 0 1 1 0 1 0 1 0 0 0 1 1 1*
1 = 1 0 1 1 1 0 0
test 130 :0 faults detected *1 0 0 1 1 0 0 1 0 0 0 1 0 1 0 1 0 1 0 1 1 0 0 0 0 1 0 1 0 1 0 0 0 0 0*
0 = 1 1 1 1 0 1 0
test 131 :0 faults detected *0 1 0 1 0 1 1 1 1 0 0 0 0 0 1 0 1 1 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 1 0*
1 = 1 1 1 0 1 0 0
test 132 :0 faults detected *1 1 0 0 1 1 0 1 1 1 0 0 1 1 1 1 0 0 0 1 1 1 0 1 0 0 0 0 0 1 1 0 1 1 1*
1 = 1 1 0 1 1 1 1
test 133 :0 faults detected *0 0 1 0 0 1 1 0 1 1 0 1 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1 0 1 1 0 1 0 1 1 1*
0 = 1 1 1 1 1 1 0
test 134 :2 faults detected *0 0 1 1 0 0 1 0 1 0 0 1 1 0 1 1 1 1 1 1 1 0 0 0 0 0 0 1 1 0 1 0 0 0 1*
1 = 0 0 0 1 1 0 0
test 135 :0 faults detected *1 1 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 0 1 1 0 0 0 1 0 0 1 1 0 0 1 0 1*
1 = 1 1 1 1 1 1 1
test 136 :0 faults detected *0 1 1 1 1 1 0 1 1 1 0 1 1 0 1 1 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 1 1 0*
0 = 1 1 0 1 0 0 1
test 137 :0 faults detected *1 0 1 0 1 0 1 1 0 0 1 1 0 1 0 1 1 1 0 0 1 0 1 0 1 0 0 0 1 0 0 0 1 0 1*
1 = 0 1 0 1 1 0 1
test 138 :0 faults detected *1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1*
0 = 1 0 1 1 0 1 0
test 139 :0 faults detected *0 1 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 0 1 0 1 0 0 1 1 0 1 1 1 1 1 1 0 1 1*
1 = 1 0 1 0 0 0 0
test 140 :0 faults detected *0 1 1 0 1 1 1 0 0 1 0 1 0 1 1 0 1 0 1 1 0 0 1 0 0 1 1 1 1 1 1 0 1 1 0*
0 = 1 1 1 1 1 1 0
test 141 :0 faults detected *0 1 1 1 0 1 0 0 0 0 1 1 1 1 1 0 1 0 1 1 0 0 0 1 0 0 0 1 1 0 0 1 1 0*
1 = 1 0 1 0 0 0 0
test 142 :0 faults detected *0 1 1 0 1 0 0 0 1 1 0 1 1 0 0 0 1 0 0 1 1 1 0 1 1 1 1 0 1 1 1 1 0 0*
1 = 1 1 0 1 1 1 0
test 143 :0 faults detected *0 0 0 1 0 0 0 0 1 1 0 1 1 0 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 0 1 0 1*
0 = 1 1 0 1 1 1 0
test 144 :1 faults detected *1 0 1 1 0 1 1 1 1 1 1 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0*
0 = 1 1 1 1 1 0 0
test 145 :0 faults detected *0 1 0 1 1 0 0 0 1 1 0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 1 0 1 1 0*
0 = 1 1 1 1 0 0 0
test 146 :0 faults detected *0 0 0 1 1 1 1 1 1 1 1 0 0 1 1 0 0 0 0 0 1 0 0 1 1 1 0 0 1 1 0 1 0 1 0 1*
0 = 1 1 1 1 1 0 1

test 147 :0 faults detected	1 1 1 0 0 1 1 0 1 1 1 1 0 1 0 1 1 0 1 1 1 0 1 0 1 1 0 1 1 1 1 0 1 1 0
0 = 1 1 1 1 0 0 0	
test 148 :0 faults detected	0 0 1 0 1 0 1 0 1 0 0 1 1 1 0 0 1 0 1 0 0 1 0 1 0 1 0 0 0 1 1 0 0 1 0
0 = 1 1 1 1 0 1 1	
test 149 :0 faults detected	0 0 1 1 1 1 0 1 1 0 0 1 0 0 1 1 0 1 1 1 1 0 0 1 0 1 0 0 0 0 0 0 1 1 0
0 = 1 1 1 1 0 0 0	
test 150 :1 faults detected	0 1 0 1 0 0 0 1 1 0 1 1 1 0 1 1 1 0 0 0 1 1 0 1 1 1 1 1 0 0 0 0 0 1 0
0 = 1 1 1 0 0 0 0	
test 151 :0 faults detected	1 1 1 0 0 0 1 0 0 1 0 0 0 0 0 0 1 1 0 1 1 1 0 1 1 0 0 1 0 1 1 1 0 0 0
1 = 1 1 1 1 1 1 0	
test 152 :0 faults detected	1 0 1 1 0 1 0 1 0 1 1 0 1 1 0 1 0 1 1 1 0 0 1 1 1 1 1 1 0 1 0 1 1 1 0
1 = 1 1 1 1 1 0 1	
test 153 :0 faults detected	1 0 1 0 0 0 0 1 0 1 0 0 1 0 1 0 0 0 1 0 0 1 0 1 0 0 0 1 0 1 1 1 1 1 0
1 = 1 1 1 1 0 1 1	
test 154 :0 faults detected	0 0 1 0 0 0 1 0 0 1 1 1 0 0 1 1 0 1 0 0 1 1 1 1 1 0 0 1 1 1 1 0 1 0 1
1 = 1 0 1 1 1 1 0	
test 155 :0 faults detected	1 0 1 1 0 1 1 0 1 1 1 0 0 1 0 1 0 1 1 1 1 0 0 1 0 1 1 0 0 1 0 0 0 1 1
0 = 1 1 1 1 1 0 1	
test 156 :0 faults detected	0 1 0 1 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
1 = 1 1 1 0 0 0 0	
test 157 :0 faults detected	1 0 0 0 1 0 0 0 0 1 1 1 0 1 1 1 1 0 0 0 1 0 0 1 0 0 0 1 1 0 1 0 0 0 1
1 = 1 0 1 1 1 1 0	
test 158 :0 faults detected	1 1 1 0 1 0 1 0 0 1 0 1 1 0 1 0 0 1 1 0 1 1 0 0 1 0 0 1 1 0 1 1 1 0 1
0 = 1 1 0 1 1 1 0	
test 159 :0 faults detected	0 0 1 1 1 1 0 1 1 1 0 0 1 1 0 1 1 1 0 0 1 0 0 0 0 0 0 1 0 0 1 1 1 0 0
0 = 1 1 0 1 1 0 1	
test 160 :0 faults detected	0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 0 0 1 0 0 1 1 0 1 0 1 1 1
1 = 1 0 0 1 0 0 0	
test 161 :0 faults detected	1 0 1 1 1 1 1 1 0 0 1 0 0 1 0 1 1 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0 1 0 0
0 = 1 1 1 1 0 1 0	
test 162 :0 faults detected	1 1 1 1 1 0 0 1 0 1 1 0 1 0 0 0 1 1 1 0 1 0 0 0 0 1 0 0 1 1 0 1 0 0 1
1 = 1 1 1 1 0 0 1	
test 163 :0 faults detected	0 1 0 1 1 1 0 0 0 1 1 1 1 1 0 0 1 0 1 0 1 0 1 0 0 1 1 1 0 0 0 0 1 0 0
1 = 1 1 0 0 0 0 0	
test 164 :0 faults detected	1 1 0 0 0 1 1 1 1 0 0 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 = 1 0 1 1 0 0 0	
test 165 :0 faults detected	1 1 0 0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 1 1 1 1 0 0 1 0 1 1 1 0 1 1 1 0 0
1 = 1 1 1 1 1 1 1	
test 166 :0 faults detected	1 1 0 1 0 1 0 1 1 1 0 1 0 1 1 1 1 0 1 1 1 1 1 1 0 0 0 0 0 1 1 0 1 1 0
1 = 1 1 0 1 0 0 0	
test 167 :0 faults detected	1 1 1 1 0 1 0 1 0 1 1 1 1 1 1 1 1 1 0 0 1 1 0 0 1 0 0 1 1 1 1 0 0 0 1 1
1 = 1 0 0 1 0 0 0	
test 168 :0 faults detected	0 0 0 1 0 0 0 1 0 0 0 1 1 0 1 0 1 0 0 0 1 0 0 1 0 1 0 1 0 1 1 0 0 1 0
0 = 1 1 1 1 0 0 0	
test 169 :0 faults detected	0 1 0 0 1 1 0 0 0 0 1 0 1 1 1 1 0 0 0 0 1 1 0 1 1 0 0 0 0 1 1 0 1 1 0
0 = 1 1 1 1 1 1 1	
test 170 :0 faults detected	1 0 0 1 0 1 1 1 1 1 0 0 0 0 0 0 1 0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 0 1 1
1 = 1 0 0 1 0 1 1	
test 171 :0 faults detected	0 1 0 1 0 0 0 1 0 1 1 1 1 0 1 1 0 0 1 1 1 0 1 0 0 0 1 0 0 0 1 0 0 0 1
0 = 1 1 1 0 0 0 0	

test 172 :0 faults detected *1 1 1 1 0 0 1 1 0 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 1 0 1 1 1 1 1 0 0 1 0*
0 = 1 1 1 1 0 0 0
test 173 :0 faults detected *0 1 1 0 0 0 1 1 0 0 1 0 1 1 0 1 0 0 1 1 0 1 1 1 0 1 1 1 0 0 0 0 0 1 0*
0 = 1 1 1 1 1 0 1
test 174 :0 faults detected *1 0 1 1 1 1 1 1 1 1 0 1 0 1 1 0 0 1 1 0 1 0 1 1 0 0 1 0 1 1 0 0 1 1 0 1*
0 = 1 1 1 1 1 0 0
test 175 :1 faults detected *1 0 0 0 0 1 1 0 0 1 0 1 1 1 0 1 0 0 0 1 0 0 1 1 0 1 0 0 0 0 1 0 1 1 0*
1 = 1 1 0 1 1 1 0
test 176 :0 faults detected *0 1 1 1 0 1 1 1 1 1 0 0 1 1 0 0 1 0 0 1 1 0 1 0 0 0 0 0 1 0 0 0 1 0 0*
1 = 1 1 0 1 1 0 1
test 177 :0 faults detected *1 0 0 0 0 0 1 1 0 0 1 1 1 1 0 0 0 0 0 0 1 0 0 1 0 1 0 0 1 1 1 1 0 1 1*
1 = 1 0 1 1 0 0 1
test 178 :0 faults detected *1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 1 0 0 0 0 1 1 1 1 1 0 1*
1 = 1 0 0 1 0 0 1
test 179 :0 faults detected *0 0 0 0 1 1 0 0 1 1 1 0 0 1 0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 0 1 0 0 0 0*
0 = 1 1 0 1 1 1 1
test 180 :0 faults detected *0 1 1 1 0 0 1 0 1 0 0 1 1 1 1 0 1 0 0 0 1 1 1 0 0 0 0 1 1 1 0 1 1 1 1*
1 = 1 0 1 0 0 1 1
test 181 :0 faults detected *0 1 0 0 0 0 1 1 0 0 1 1 0 1 0 1 0 1 0 1 0 0 0 1 1 0 1 0 1 1 0 1 1 1 0*
1 = 1 1 1 0 0 0 0
test 182 :0 faults detected *0 0 0 0 1 0 1 1 0 1 0 0 1 1 1 0 0 0 0 0 0 0 0 1 0 1 1 1 0 0 1 1 0 0 1*
1 = 1 0 1 1 1 0 1
test 183 :0 faults detected *0 0 0 1 1 1 1 0 0 0 0 1 1 0 1 1 0 0 0 0 1 0 0 1 0 1 1 0 0 0 1 0 0 0 1*
0 = 0 0 1 1 1 1 1
test 184 :0 faults detected *0 1 0 0 0 1 1 0 0 0 1 1 0 1 0 1 1 1 1 0 1 0 1 0 1 0 0 1 1 1 0 1 1 1 1*
1 = 1 1 1 0 0 0 0
test 185 :0 faults detected *0 0 0 1 1 1 1 0 1 1 0 0 1 1 0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 0 1 1 0 1*
0 = 1 1 1 1 1 0 1
test 186 :0 faults detected *1 1 0 1 0 0 1 0 0 0 1 0 1 1 1 0 0 1 0 0 1 1 1 0 0 1 0 1 0 1 0 0 0 1 1*
1 = 1 1 1 1 0 1 0
test 187 :0 faults detected *1 1 0 0 1 1 0 1 1 0 0 0 1 1 1 1 1 1 0 1 0 1 0 0 1 0 0 0 1 1 1 1 0 1 0*
0 = 1 1 0 1 1 1 1
test 188 :0 faults detected *0 1 0 1 0 0 1 1 1 0 0 0 1 0 0 1 1 0 0 1 1 0 1 0 0 1 0 1 0 0 1 1 0 0 1*
1 = 1 1 1 0 0 1 0
test 189 :0 faults detected *1 1 0 1 1 0 0 0 1 0 0 0 1 1 0 1 0 1 1 1 0 0 1 0 0 0 0 1 0 0 1 0 0 0 1*
1 = 1 1 1 1 1 0 1
test 190 :0 faults detected *0 0 1 1 0 0 1 1 1 1 1 0 0 1 0 0 0 0 1 0 1 0 1 0 0 1 1 0 0 1 0 0 1 1 1*
1 = 1 1 1 1 1 0 1
test 191 :0 faults detected *0 0 1 1 0 0 0 0 0 1 1 1 1 0 1 0 1 1 1 0 1 1 1 0 0 1 0 1 0 0 1 1 1 1 0*
0 = 1 1 1 1 0 1 0
test 192 :0 faults detected *1 0 0 1 1 0 1 1 1 0 0 0 1 0 1 0 1 1 0 0 1 0 0 1 1 0 1 0 1 0 0 0 1 1 1*
1 = 1 1 0 1 1 0 0
test 193 :0 faults detected *1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 1 0 1 1 0 1 1 0 1 1 1 1 0 0 0 1 1 0 1 0*
1 = 1 1 0 1 1 1 0
test 194 :1 faults detected *0 1 1 0 0 0 1 1 0 0 0 1 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 1 1 1 0 0 0 1 1*
0 = 1 1 0 1 0 1 0
test 195 :0 faults detected *1 0 1 1 0 1 1 1 0 1 1 0 0 1 1 0 0 0 1 1 0 0 0 0 1 1 1 1 0 1 0 1 0 0 1*
1 = 1 0 1 1 1 0 1
test 196 :0 faults detected *0 1 1 1 1 1 0 0 0 0 1 1 0 0 1 0 0 0 0 1 1 1 1 0 1 1 0 0 0 1 1 1 0 0 0*
1 = 1 1 1 1 0 1 0

```
test 197:0 faults detected        1 1 1 1 0 1 0 1 0 0 0 1 0 0 0 0 1 0 0 0 1 1 1 0 0 0 0 0 1 0 0 0 1 0 1
0 = 1 0 1 1 0 1 1
test 198:0 faults detected        0 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 0 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 0 1 1
1 = 1 1 1 0 0 0 0
test 199:0 faults detected        1 1 1 0 0 0 0 1 0 1 0 1 1 0 1 0 1 1 0 1 0 0 1 1 0 0 1 0 1 0 0 1 1 1 0
1 = 1 1 1 1 1 0 0

End of fault simulation.

*******************************************************
*                                                     *
*       Welcome to Digital Circuit Verification System*
*                                                     *
*       Writer : Liwu Jin              Systems Science*
*                                                     *
*       University of Ottawa    Copyright (C) 2003.11 *
*                                                     *
*******************************************************

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

1.      Circuit Structure
        Name of the circuit                 :   C432
        Number of gates                     :   196
        Number of primary inputs            :   36
        Number of primary outputs           :   7
        Depth of the circuit                :   153

2.      Simulation parameters
        Simulation mode                     :   random
        Initial random number generator seed:   1050420308

3.      Simulation results
        Number of test patterns applied     :   199
        Fault coverage                      :   95.918 %
        Number of collapsed faults          :   392
        Number of detected faults           :   376
        Number of undetected faults         :   16

4.      Total running time                  :   1076 seconds
        Simulation time                     :   430 seconds
```